Linda®

User Guide

September, 2005

SCIENTIFIC
COMPUTING ASSOCIATES
INC.

Acknowledgments

Contents

Introduction

This manual describes the Linda® product family—TCP Linda, the Linda Code Development System (CDS), and Shared Memory Linda (SM Linda). The Linda product family are parallel programming languages based on the C and Fortran languages that enable users to create parallel programs that perform well in a wide range of computing environments.

This manual includes examples of using both the C-based and Fortran-based Linda products and refers to them as "C-Linda" and "Fortran-Linda." C-Linda combines the coordination language Linda with the programming language C; similarly, Fortran-Linda combines the coordination language Linda with the Fortran programming language. Parallel programs are created with Linda by combining a number of independent computations (processes) into a single parallel program. The separate computations are written in C or Fortran, and Linda provides the glue that binds them together.

Linda has several characteristics which set it apart from other parallel programming environments:

- Linda augments the serial programming language (C or Fortran). It does not replace it, nor make it obsolete. In this way, Linda builds on investments in existing programs.

- Linda parallel programs are portable, and they run on a large variety of parallel computer systems, including shared-memory computers, distributed memory computers, clusters, and networks of computers. With few exceptions, Linda programs written for one environment run without change on another.

- Linda is easy to use. Conceptually, Linda implements parallelism via a logically global memory (virtual shared memory), called *tuple space*, and a small number of simple but powerful operations on it. Tuple space and the operations that act on it are easy to understand and quickly mastered. In addition, the C-Linda and Fortran-Linda compilers support all of the usual program development features, including compile-time error checking and runtime debugging and visualization.

How to Use this Manual

If you are new to parallel programming, Chapter 1: Overview of Parallel Programming and the Linda Model, provides a general introduction to the subject and to Linda. It also introduces tuple space, the Linda operations, and other essential concepts.

For a more in depth look at parallel programming and algorithm development, SCIENTIFIC recommends the book, *How to Write Parallel Programs: A First Course*, by Nicholas Carriero and David Gelernter, which is available in PDF format at no cost on the SCAI Website at www.lindaspaces.com. See the Bibliography for the complete citation for this book and related works.

If you are using Linda to create parallel programs, Chapter 2: Using the Linda Operations, describes the Linda operations in detail. It explains the program compilation and execution process, provides a couple of simple example programs, and includes an extended discussion of tuple-matching rules and restrictions.

If you are using TCP Linda, Chapter 3: Using TCP Linda, describes the special features and considerations of the Linda implementation for networks of UNIX workstations. The first section provides an introduction and "quick start" for new users of TCP Linda. The remainder of the chapter describes the general features of TCP Linda.

To see examples of using Linda with the C and Fortran programming languages, Chapter 4: Case Studies, presents several extended program examples illustrating the process of transforming a sequential program into a parallel program with Linda.

Chapter 5: Using Tuplescope, describes the Tuplescope visualization and debugging tool and the Linda Code Development System (CDS) and how to use them for high-level debugging of Linda programs running on a single node.

Chapter 6: Linda Usage and Syntax Summary, provides a programmer's reference to the features of the Linda programming environment, including both C-Linda and Fortran-Linda language constructs and the elements of the Linda Toolkit.

Chapter 7: Sample Programs, contains five sets of example programs that demonstrate how to use Linda to parallelize serial programs. Each example includes the serial version of a program written in C and a parallel version created with Linda. The examples are presented in order of increasing complexity.

The Bibliography lists books and articles that may be of interest to Linda users. Some items provide more advanced treatment of parallel programming techniques, while others discuss the example programs in greater detail or from different perspectives.

About the Example Programs

The chapters in this guide that describe using Linda include many code samples and fragments as examples. All of this code is derived from real programs, but in most cases has been shortened and simplified, usually to make it fit into the allowed space. Typically, declarations and preprocessor directives are omitted except when they are vital to understanding the program. Also, sections of code that are not relevant to the point being made are replaced often by a one-line summary of their function (set in italics). Blank lines (without initial comment indicator) have been inserted into Fortran programs for readability. Therefore, although the code examples are derived from real programs, they do not generally constitute "working" code. Many examples are provided in both C

and Fortran versions; differences between the two languages are highlighted in the text where appropriate. For examples of "working code," see the chapter, <u>Sample Programs</u> on page 127.

Typographic Conventions

`Fixed-width` type is used for all code examples, whether set off in their own paragraphs or included within the main text. For example, variable names and filenames referred to within the text are set in fixed-width type.

`Boldface fixed-width` type is used in examples to indicate text—usually commands to the operating system—typed by the user.

Italic type is used for replaceable arguments in operation and command definitions, for summary lines and other descriptions within example code, and occasionally for emphasis.

Boldface sans-serif type is used for Linda operation names used in a generic way within normal text, for non-varying text within syntax definitions, and for menu buttons and other controls in <u>The Tuplescope Display</u>.

Italic sans-serif type is used for replaceable arguments within formal syntax definitions and when they are referred to within the text.

1
Overview of Parallel Programming and the Linda Model

People always are trying to make programs run faster. One way to do so is to divide the work the program must do into pieces that can be worked on at the same time (on different processors). More formally, creating a parallel program depends on finding independent computations that can be executed simultaneously.

Producing a parallel program generally involves three steps:

* Developing and debugging a sequential program. For existing programs, this step is already done.
* Transforming the sequential program into a parallel program.
* Optimizing the parallel program.

Of course, if the second step is done perfectly then the third one will be unnecessary, but in practice that rarely happens. It's usually much easier—and quicker in the long run—to transform the serial program into a parallel program in the most straightforward way possible, measure its performance, and then search for ways to improve it.

If we focus on step 2, a natural question arises: where does the parallelism come from? The language of parallel programming can be quite ambiguous. On the one hand, there is talk of "parallelizing programs," a phrase which focuses on the programmers who convert sequential programs into parallel ones. On the other hand, much is said about "exploiting parallelism," implying that parallelism is already inherent in the program itself. Which is correct? Is it something you find or something you create?

The answer is, of course, "it depends." Sometimes a program can be trivially adapted to run in parallel because the work it does naturally divides into discrete chunks. Sometimes a serial program must be restructured significantly in order to transform it into a parallel program, reorganizing the computations the program does into units which can be run in parallel. And sometimes an existing program will yield little in the way of independent computations. In this case, it is necessary to rethink the approach to the problem that the program addresses, that is create new algorithms to formulate a solution which can be implemented as a parallel program.[†]

[†] Of course, even in these cases, it may still be possible to take advantage of a parallel computer by running multiple concurrent sequential jobs.

To put it another way, a computer program doesn't merely solve a particular problem, but rather it embodies a particular approach to solving its specific problem. Making a parallel version potentially can involve changes to the program structure, or the algorithms it implements, or both. The examples in this manual include instances of all three possibilities.

Once the work has been divided into pieces, yielding a parallel program, another factor comes into play, the inherent cost associated with parallelism, specifically the additional effort of constructing and coordinating the separate program parts. This overhead often is dominated by the communication between discrete program processes, and it increases naturally with the number of chunks the program is divided into, eventually reaching a point of diminishing return in which the cost of creating and maintaining the separate execution threads overshadows the performance gains realized from their parallel execution. An efficient parallel program will maximize the ratio of work proceeding in parallel to the overhead associated with its parallel execution.

This ratio of computation to communication is referred to as *granularity*. Think of dividing a rock into roughly equal-sized parts. There are many ways to do it; in fact, there is a continuum of possibilities ranging from fine grains of sand at one end to two rocks half the size of the original at the other. A parallel program that divides the work into many tiny tasks is said to be fine-grained, while one that divides the work into a small number of relatively large ones is called coarse-grained.

There is no absolute correct level of granularity. Neither coarse-grained nor fine-grained parallelism is inherently better or worse than the other. However, when overhead overwhelms computation, a program is too fine grained for its environment, whatever its absolute granularity level may be. The optimum level depends on both the algorithms a program implements and the hardware environment it runs in. A level of granularity that runs efficiently in one environment (for example, a parallel computer with very fast interprocessor communication channels) may not perform as well in another (such as a network of workstations with much slower communication between processors).

There are two ways to address this issue (and we'll look at examples of both of them in the course of this manual). First, many problems offer a choice of granularity level. For example, if a program must execute eight independent matrix multiply operations, a parallel program could perform all eight of them at the same time, or execute eight parallel matrix multiplication operations one after another. Which approach is correct depends on the structure of the overall problem, and each is better than the other in some circumstances.

The other solution is to build adjustable granularity into programs so that they can be easily modified for different environments. Changing the granularity level then becomes as simple as changing a few parameter definitions. This technique complements the preceding one, and both can be used in the same program.

These are the major issues facing any parallel programmer. In the next section we'll look at three different approaches to creating parallel programs and indicate how Linda is situated with respect to each of them.

Approaches to Parallel Programming

There are two main challenges facing any parallel programmer:

- How to divide the work among the available processors.[‡]
- Where to store the data and how to get it to processors that need it.

Two radically different approaches to these problems have emerged as the dominant parallel processing paradigms. They are message passing and distributed data structures implemented in virtual shared memory.

Message Passing

Message passing focuses on the separate processes used to complete the overall computation. In this scheme, many concurrent processes are created, and all of the data involved in the calculation is distributed among them in some way. There is no shared data. When a process needs data held by another one, the second process must send it to the first one.

For example, let's again consider a matrix multiplication calculation:

```
A * B = C
```

A message passing version might create many processes, each responsible for computing one row of the output matrix C. If the matrices are large enough, it might not be possible for each process to hold all of B at once. In this case, each process might then hold the row of A, corresponding to its output row, and, at any given time, one column of B. The process computes the dot product of its row and the column it currently holds, producing one element of its output row. When it finishes with one column, it sends it on to another process, and receives a new column from some process. Once all processes have received all the columns of B and have finished their final dot product, the matrix multiplication is complete (although the completed rows would still need to be explicitly moved from the component processors to the desired output location).

The message passing approach to the problem is illustrated in the diagram below. It has a number of implications for the programmer. First, the program needs to keep track of which process has what data at all times. Second, explicit send data and receive data operations must be executed whenever data needs to move from one process to another. Unless they are coded extremely carefully, such bookkeeping and communication activities can cause bottlenecks in program execution.

[‡] The term *processor* is used in a generic sense here to designate a distinct computational resource whether it is one CPU in a multiprocessor computer or a separate computer on a network.
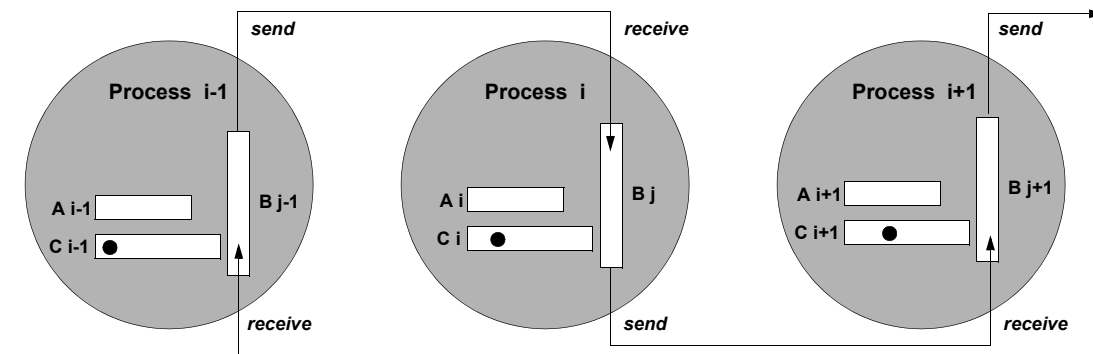
Message Passing Matrix Multiplication

In message passing parallel programs, there is no global data. All data is always held by some process and must be explicitly sent to other processes that need it. Here each process holds one row of A and computes one element of C while it has the corresponding column of B. The columns of B are passed from process to process to complete the computation.

Distributed Data Structures

Distributed data structure programs are a second approach to parallel programming. This method decouples the data required for the calculation and the distinct simultaneously-executing processes which each perform a part of it, making them autonomous. Distributed data structure programs use a shared data space, with the individual processes reading data from it and placing results into it. The data structure is distributed in the sense that different parts of it can reside in different processors, but it looks like one single global memory space to the component processes. In this sense, a distributed data structure might be termed virtual shared memory (VSM). Although Linda can be used to implement both types of parallel programs, the most natural one for use with Linda is the distributed data structures using virtual shared memory.

All interprocess communication is accomplished via this global data space. Processes never explicitly send messages to one another, but rather place data into the shared data space. When another process needs that data, it obtains it from the shared data space. Linda handles all data transfer operations, so the program need not worry about the exact mechanism by which it occurs. Linda operations require no additional overhead over any message passing scheme, and in fact sometimes are more efficient!

Programs using the distributed data structure method often use a master/worker computation strategy. Under this approach, the total work to be done by the program is broken into a number of discrete tasks which are stored in the global data space. One process, known as the *master*, is responsible for generating the tasks and gathering and processing the results. Actual program execution involves a number of component processes known as *workers*. Each worker removes a task, completes it, and then grabs another, continuing until some condition is met. For example, it may encounter a special type of task—known as a *poison pill*—telling it to die, or it may be terminated by some other mechanism. Depending on the situation, the master process may also perform task computations in addition to its other duties.
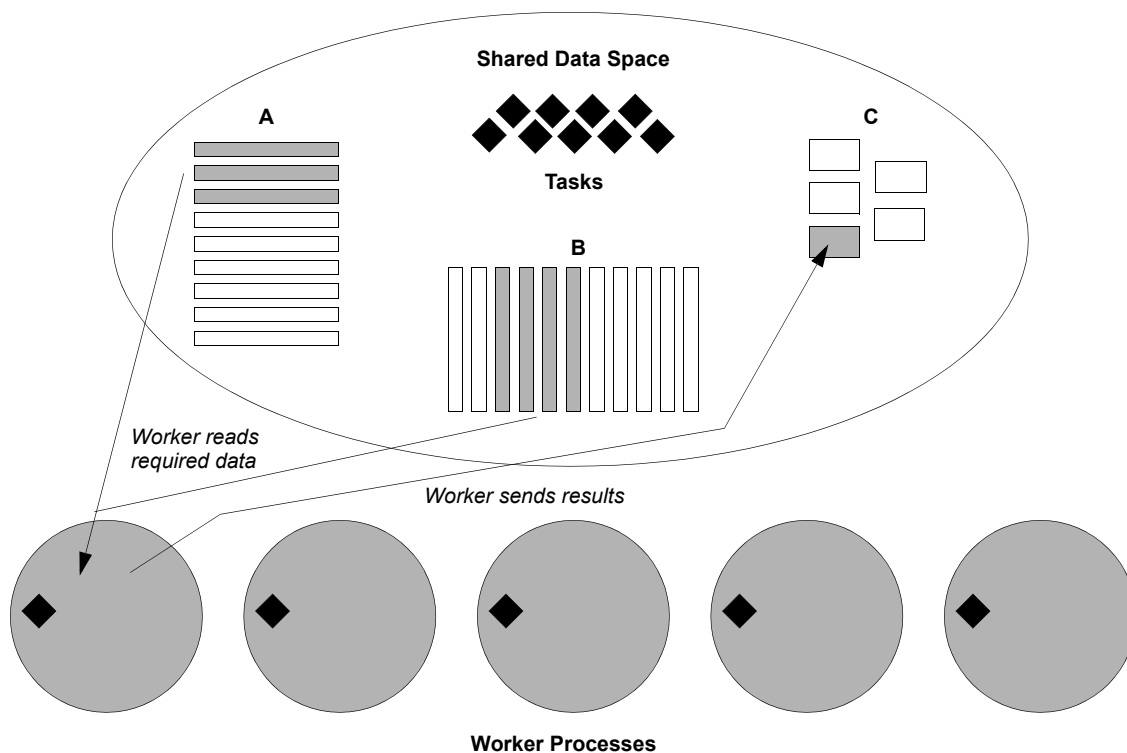
Distributed Data Structures Matrix Multiplication

In a distributed data structures parallel program, workers retrieve tasks from the shared data space, complete them, and then repeat the process until all tasks are done. Here each task is to compute a portion of the result matrix C. Each worker reads the data it needs for its current task—here the relevant portions of A and B—from the global data space, and places its results there when finished as well. The master process, which generated the tasks and placed them into the shared data space, also eventually gathers the results placed there by the worker processes.

Note that the tasks and workers are also independent of one another. The total work is not split among the workers; rather, the total work is split into a number of chunks, and each worker performs task after task until the entire job is done. In this case, it is the task size, and not merely the number of workers, that primarily determines the granularity of the program, and this granularity can be adjusted by varying the task size.

If we look again at our matrix multiplication example, each task might consist of computing some part of the output matrix C. At the beginning of the program, the master process creates tasks for each chunk of C that is to be computed separately. Each worker removes a task from the shared data space. It then reads the required rows of A and columns of B (if necessary), and forms the dot products. When it is finished, it places the resulting chunk of C into the shared data space, and at the conclusion of the program, the master gathers up all the chunks of C. The granularity of the calculation can be adjusted by varying the amount of C that each task computes. This approach to the problem is illustrated in the preceding diagram.

Notice once again the distinction between tasks and workers. In this example, the elements of C to be computed are divided into groups, and each task consists of computing one of the groups. The elements of C are not divided among the worker

processes in any explicit way. Worker processes do not know what tasks they will perform when they are created. Workers acquire tasks as they are ready, and perform whatever task they get.

This approach has many benefits. For one thing, it is generally easy to code, since the worker processes don't need to worry about explicit interprocess communication; that is taken care of by the parallel programming environment, which manages the shared data space. Processes read and write data via Linda's operations. In addition, this method also tends to be naturally load balancing. Workers continually execute tasks as long as any of them remain. If one worker runs on a faster processor than some others, it will finish each task more quickly and do proportionately more of them. (Of course, there are times when reality isn't quite this simple, so we'll look at some techniques to ensure good load balancing in Chapter 3.)

The Linda Model

Linda—or more precisely, the Linda model—is a general model of parallel computing based on distributed data structures (although as we've noted before, it can be used to implement message passing as well). Linda calls the shared data space *tuple space*. C-Linda is an implementation of the Linda model using the C programming language, and Fortran-Linda is an implementation of the Linda model using the Fortran programming language. Processes access tuple space via a small number of operations that Linda provides. For example, parallel programs using C-Linda are written in C and incorporate these operations as necessary to access tuple space. In this way, C-Linda functions as a coordination language, providing the tools and environment necessary to combine distinct processes into a complete parallel program. The parallel operations in C-Linda are orthogonal to C, providing complementary capabilities necessary to parallel programs. C-Linda programs make full use of standard C for computation and other non-parallel tasks; C-Linda enables these sequential operations to be divided among the available processors. Since C-Linda is implemented as a precompiler, C-Linda programs are essentially independent of the particular (native) C compiler used for final compilation and linking. Fortran-Linda operates in an analogous manner.

Linda programmers don't need to worry about how tuple space is set up, where it is physically located, or how data moves between it and running processes; all of this is managed by the Linda software system. Because of this, Linda is logically independent of system architecture, and Linda programs are portable across different architectures, whether they're shared memory computers, distributed memory computers, or networks of workstations.

Data moves to and from tuple space as tuples.[††] Tuples are the data structures of tuple space. A tuple is a sequence of up to 16 typed fields; it is represented by a comma-separated list of items enclosed in parentheses. Here is a simple example:

C Form   **Fortran Form**
("simple", 1)   ('simple', 1)

[††] Pronounced "two-pull" with the emphasis on the first syllable.

This tuple has two fields; the first is a character string, and the second is an integer, and in this case, both of them contain literal values. Variables may also be used in tuples:

C Form

```
("easy", i)
```

Fortran Form

```
('easy', i)
```

This tuple also has a string as its first field, and a second field of whatever type the variable `i` is. The value in the second field is `i`'s current value.

Linda provides four operations for accessing tuple space (plus two variant forms described in <u>Predicate Operation Forms: inp and rdp</u> on page 25):

Operation	*Action*
out	Places a data tuple in tuple space.
eval	Creates a live tuple (usually starting new process(es)).
in	Removes a tuple from tuple space.
rd	Reads the values in a tuple in tuple space, leaving the tuple there.

For example, this **out** operation places a data tuple with one string and two integer fields into tuple space:

C Form

```
out("cube", 4, 64);
```

Fortran Form

```
out('cube', 4, 64)
```

There are two kinds of tuples: *data tuples* (also called *passive tuples*), like those we've looked at so far, which contain static data, and *process tuples*, also known as *live tuples*, which are under active evaluation.

An **eval** operation creates a process tuple consisting of the fields specified as its argument and then returns. This process tuple implicitly creates a process to evaluate each argument. Actual implementations create a process only for arguments consisting of a simple function call (and fulfilling some other conditions which we'll note later); all other fields in the **eval** are evaluated sequentially.

While the processes run, the **eval**'s tuple is referred to as a live tuple; as each process completes, its return value is placed into the corresponding field, and once all fields are filled—all processes have completed—the resulting data tuple is placed into tuple space. While it is not literally true that an **eval** creates the processes that evaluate its arguments, it can be helpful to think of it this way.

For example, this **eval** statement will result in a process being created to evaluate its third argument, `f(i)`:

C Form

```
eval("test", i, f(i));
```

Fortran Form

```
eval('test', i, f(i))
```

evals are often used to initiate worker processes, as in the following loop:

C Form	Fortran Form

```
for (i=0;i < NWORKERS;i++)     Do 5 I=1,NWORKERS
   eval("worker", worker());      eval('worker', worker())
                               5 Continue
```

This loop starts NWORKERS worker processes. In this case, the primary function of the **eval** is simply to start the process, rather than to perform a computation and place the result into tuple space. Formally, however, when each worker finishes, a tuple of the form:

C Form	Fortran Form

```
("worker", 0)                  ('worker', 0)
```

is placed into tuple space (assuming that the workers terminate normally and adhere to the usual UNIX return value convention of zero for success).

The other two operations allow a process to access the data in tuple space. A **rd** operation reads a tuple from tuple space, and an **in** operation removes a tuple from tuple space.

Both **rd** and **in** take a *template*‡‡ as their argument. A template specifies what sort of tuple to retrieve. Like tuples, templates consist of a sequence of typed fields, some of which hold values (such fields are known as *actuals*)—either constants or expressions which resolve to constants—and some of which hold placeholders for the data in the corresponding field of the matched tuple in tuple space. These placeholders begin with a question mark and are known as *formals*. When a matching tuple is found, variables used as formals in the template will be assigned the values in corresponding fields of the matched tuple.

Here is an example:

C Form	Fortran Form

```
("simple", ?i)                 ('simple', ?i)
```

In this template, the first field is an actual, and the second field is a formal. If this template is used as the argument to a **rd** operation, and a matching tuple is found, then the variable i will be assigned the value in the second field of the matched tuple.

A template matches a tuple when:

* They both have the same number of fields.
* The types, values, and length of all actuals (literal values) in the template are the same as the those of the corresponding fields in the tuple.
* The types and lengths of all formals in the template match the types and lengths of the corresponding fields in the tuple.

We'll consider these conditions in more detail in Chapter 2; for now, let's look at some examples. If the tuple:

---

‡‡ Also known as an *anti-tuple*.

C Form

```
("cube", 8, 512)
```

Fortran Form

```
('cube', 8, 512)
```

is in tuple space, then the statement:

C Form

```
rd("cube", 8, ?i);
```

Fortran Form

```
rd('cube', 8, ?i)
```

will match it and assign the value 512 to `i`, assuming that `i` is an integer. Similarly, if `j` is an integer variable equal to 8, then the statement:

C Form

```
in("cube", j, ?i);
```

Fortran Form

```
in('cube',j, ?i)
```

will match it (again if `i` is an integer), assign the value 512 to `i`, and remove the tuple from tuple space.

If more than one matching tuple for a template is present in tuple space, one of the matching tuples will be used. Which one is non-deterministic; it will not necessarily be the oldest, the most recent, or a tuple specified by any other criteria. Programs must be prepared to accept any matching tuple, and to receive equivalent tuples in any order. Similarly, repeated **rd** operations will often yield the same tuple each time if the tuples in tuple space remain unchanged.

If no matching tuple is present in tuple space, then both **rd** and **in** will wait until one appears; this is called *blocking*. The routine that called them will pause, waiting for them to return.

Note that the direction of data movement for the **in** and **out** operations is from the point of view of the process calling them and *not* from the point of view of tuple space. Thus, an **out** places data into tuple space, and an **in** retrieves data from it. This is similar to the general use of input and output in conventional programming languages.

All data operations to and from tuple space occur in this way. Data is placed into tuple space as tuples, and data is read or retrieved from tuple space by matching a tuple to a template, not by, say, specifying a memory address or a position in a list. This characteristic defines tuple space as an *associative memory model.*

The examples so far have all used a string as the first element of the tuple. This is not required but is a good practice to adopt in most cases because it makes Linda programs more readable and easier to debug. It also helps the Linda compiler to easily separate tuples into discrete classes enabling more efficient matching.

The next chapter will look at more complicated tuples and tuple matching scenarios and will present some simple examples of parallel programs using Linda.

How and Where to Parallelize

This section contains a short discussion of how to find the computational kernels in a program. It discusses the UNIX profiling utilities **prof** and **gprof**. The steps described here are independent of Linda and are usually done before parallelizing the program. They are designed to help you determine where to focus your efforts within the original application. This section by nature is introductory and brief; consult the relevant manual pages and related works in the Bibliography for more detailed discussions of these commands and profiling in general.

To use the UNIX profiling utilities, the **-p** (for **prof**) or **-pg** (for **gprof**) options must be included on the link command. Note that they are not needed for compilations, only for linking. For example, the following command prepares the program `test24` for use with **gprof**:

```
$ cc -o test24 -pg test24.o
```

Then, you run the resulting executable in the normal manner. Doing so will create a file named `mon.out` (**prof**) or `gmon.out` (**gprof**) in the directory from which the program was executed. These files contain the profiling data obtained during the run. You then run **prof** or **gprof** on the output files.

There can be a lot of output from both of these commands. Among the most useful are the breakdown of time spent, the number of times each routine was called, and the call graph information (where each routine was called from). Here is an example of the first:

```
 %time    seconds   cum %   cum sec   procedure(file)

 29.2     235.9100  29.2    235.91    gaus3_ (gaus3.f)
 24.6     198.5800  53.8    434.49    dgemm_mm_ (dgemm_mm.s)
 13.0     105.1600  66.8    539.65    func3_ (func3.f)
  9.1      73.2500  75.8    612.90    tria_ (tria.f)
  8.0      64.8500  83.9    677.75    exp (exp.s)
  7.2      58.5500  91.1    736.30    intarc_ (intarc.f)
...
```

This display shows the total amount and percentage of CPU time used by each routine, in decreasing order. In this program, 90% of the total execution time is spent in just 6 routines, one of which is a matrix multiply library call. About 8% of the time is spent in calls to the exponential function.

The following display is an example of a call frequency table:

```
 calls      %calls   cum%    bytes   procedure (file)

 20547111   68.53    68.53   480     exp (exp.s)
      768   0.00     68.54   17072   gaus3_ (gaus3.f)
...
```

This sort of display summarizes the number of times a given routine was called. Often, it is helpful to also know where a routine was called from. A call graph table will indicate this information. Here is an example:

```
called procedure #calls    %calls    from line  calling proc(file)

exp               7557120  36.78      48         gaus3_ (gaus3.f)
                  3022848  14.71      63         gaus3_ (gaus3.f)
                  3022848  14.71      79         gaus3_ (gaus3.f)
                  3022848  14.71      95         gaus3_ (gaus3.f)
                   503808   2.45     143         gaus3_ (gaus3.f)
                   503808   2.45     127         gaus3_ (gaus3.f)
                   503808   2.45     111         gaus3_ (gaus3.f)
                   503808   2.45     159         gaus3_ (gaus3.f)
                   503808   2.45     175         gaus3_ (gaus3.f)
                   503808   2.45     191         gaus3_ (gaus3.f)
sqrt              1007616  15.03     111         func3_ (func3.f)
                  1007616  15.03     110         func3_ (func3.f)
                  1007616  15.03     108         func3_ (func3.f)
                  1007616  15.03     109         func3_ (func3.f)
                   503808   7.51      44         func3_ (func3.f)
                   503808   7.51     147         func3_ (func3.f)
                   503808   7.51     148         func3_ (func3.f)
                   503808   7.51     149         func3_ (func3.f)
...
```

Here we can easily see that the exponential function is called literally millions of times, all from within one routine. We would want to try to do some of those calls in parallel if they are independent.

2

Using the Linda Operations

In the last chapter we discussed the process of creating parallel programs from a general point of view. We also introduced tuples and the main Linda operations. In this chapter we will look at them in more detail and in the context of complete, if simple, parallel programs. Note that this chapter contains examples of using both C-based and Fortran-based Linda, and it refers to them as "C-Linda" and "Fortran-Linda" respectively.

Quick Start: Hello, world

In this section we'll construct a parallel version of the canonical first example program, **hello_world**. Here is the sequential version:

C:
```c
main()
{
   printf("Hello, world\n");
}
```

Fortran:
```fortran
      Program Hello_World
      Print *, 'Hello, world'
      End
```

It would be absurd to try to perform the "computation" done by this program in parallel, but we can create a parallel version where each worker process executes this program— and says "Hello, world"—at the same time. Here is a program that does so:

C:
```c
real_main(int argc,char *argv[])
{
   int nworker, j, hello();
   nworker=atoi(argv[1]);

   for (j=0; j<nworker; j++) eval("worker", hello(j));
   for (j=0; j<nworker; j++) in("done");
   printf("hello_world is finished\n");

   return(0);
}
```

Fortran:
```fortran
      Subroutine real_main
      Integer I, NProc
```

```
          Obtain number of workers & store in NProc
          Do 10 I=1,NProc
            eval('worker', hello(I))
 10    Continue
          Do 11 I=1,NProc
            in('done')
 11    Continue
          Print *,'hello_world is finished'
          Return
          End
```

The first thing to notice about this program is its name and top-level structure:
real_main for the C-Linda, which requires that the top level routine be given this name
rather than the usual main. Similarly, the top-level Fortran-Linda routine is named
real_main; note also that it is defined as a Subroutine, not as a Program.

The C-Linda program requires one command line argument: the number of worker
processes to create (and to save space we've eliminated the code that checks whether or
not it got a valid argument). There are a number of ways that the Fortran-Linda version
could obtain the number of worker processes; we won't dwell upon those possibilities at
this point.

Next, the program's first loop initiates nworker worker processes using the **eval**
operation, each executing the function hello. Here is hello:

C:
```
hello(int i)
{
    printf("Hello, world from number %d\n",i);
    out("done");
    return(0);
}
```

Fortran:
```
          Subroutine Hello(ID)

          Print *,'Hello, world from number ',ID
          out('done')
          Return
          End
```

hello is only a minor variation of our original sequential program. It prints the "Hello,
world" message along with the number it was passed from real_main (this integer
serves as a sort of internal process number). Each message will look something like this:

```
Hello, world from number 3
```

The routine hello places a tuple containing the string "done" into tuple space just
before exiting. These tuples are then gathered up by the master process—real_main—
in its second loop. This technique has the effect of forcing real_main to wait until all
the worker processes have terminated before exiting itself, a recommended Linda

programming practice. Each **in** operation removes one "done" tuple from tuple space, and it will block if none is currently present and wait until some worker finishes and sends its tuple there.

This same effect could have been achieved by means of a counter tuple which each worker process increments as it finishes. In this case, `real_main` would create and initialize the counter:

C Version	**Fortran Version**
`out("counter", 0);`	`out('counter', 0)`

and each worker would update it as its last action:

C Version	**Fortran Version**
`in("counter", ?j);`	`in('counter', ?J)`
`out("counter", j+1);`	`out('counter', J+1)`

These statements remove the counter from tuple space, assign the current value of its second field—the number of processes that have finished so far—to the variable `j`, and then increment it and place the tuple back into tuple space. Note that only one process can access the counter tuple at a time, and so some processes may have to wait for others to finish before they can terminate. In this case, the waiting time is minuscule, so for this program, the concern is a non-issue. However, in general it is best to avoid building unnecessary dependencies among processes into a program.

With a counter tuple, the final loop in `real_main` would be replaced by the statement:

C Version	**Fortran Version**
`in("counter", nworker);`	`in('counter', NProc)`

`real_main` will block until the counter tuple's second field has its final value, the number of worker processes.

A third approach involves retrieving the final data tuples created after the **eval**'ed processes exit, for example:

C Version	**Fortran Version**
`in("worker",?retval);`	`in('worker', ?iretval)`

This allows the program to examine the return code from the function started by the **eval** operation. While it isn't really necessary for a function as simple as `hello`, it is a technique that is quite useful in more complex programs.

Compiling and Running the Program

To run this program, you must first compile and link it. The C-Linda compiler has similar syntax to standard C compilers. Its name is **clc**, and its source files must have the extension `.cl`. Here is a **clc** command that would compile the program in the file `hello_world.cl`:

C-Linda Compilation

```
$ clc -o hello_world hello_world.cl
```

The **-o** option has the same meaning as with other compilers, so this command would compile and link the source file `hello_world.cl`, creating the executable program `hello_world`.

The Fortran-Linda version of the program is created using the **flc** command, as shown below.

Fortran-Linda Compilation

```
$ flc -o hello_world hello_world.fl do_args.f
```

Note that the extension on the Fortran-Linda source file is `.fl`. As illustrated, additional source files (Linda and non-Linda alike) may also be included on a Linda compilation command line when appropriate.

Here is a sample run:

Program Execution

```
$ hello_world 8
Hello, world from number 3.
Hello, world from number 1.
Hello, world from number 0.
Hello, world from number 4.
Hello, world from number 5.
Hello, world from number 2.
Hello, world from number 7.
Hello, world from number 6.
hello_world is finished.
```

It is to be expected that the messages from the various processes will display in non-numerical order since we've done nothing to force them to display sequentially. Linda programs are essentially asynchronous, and there is no guarantee that a particular process will execute before any other. Indeed, we would not want to do so, since we're trying to achieve a simultaneously executing parallel version of **hello_world**.

To run on a network, the program is compiled and linked in essentially the same way, but running it requires using a slightly different command. For this version, we might want to add the node name to the output line from each worker:

C:

```
gethostname(name,20);
printf("Hello, world from number %d running on %s.\n",i,name);
```

Shown below are the commands to create and run the modified program (C-Linda version is shown):

**Program
Execution with
TCP Linda**

```
$ clc -o hello_world hello_world.cl
$ ntsnet hello_world 8
Hello, world from number 4 running on moliere.
Hello, world from number 2 running on ibsen.
Hello, world from number 0 running on cervantes.
Hello, world from number 7 running on sappho.
Hello, world from number 3 running on blake.
Hello, world from number 1 running on virgil.
Hello, world from number 6 running on leopardi.
Hello, world from number 5 running on goethe.
hello_world is finished.
```

The **ntsnet** command initiates a Linda program on a network. **ntsnet** and its configuration and options are described in detail in <u>Using TCP Linda</u>.

Linda Operations

This section describes the Linda operations we looked at in the previous chapter in more detail, including some simple examples. Additional examples are given in the next section, "Tuples and Tuple Matching."

in

The **in** operation attempts to remove a tuple from tuple space by searching for a data tuple which matches the template specified as its argument. If no matching tuple is found, then the operation blocks, and the process executing the **in** suspends until one becomes available. If there are one or more matching tuples, then one of them is chosen arbitrarily. The matching tuple is removed from tuple space, and each formal in the template is set to the value of its corresponding field in the tuple.

For example, this **in** operation removes a tuple having the string "coord" as its first field and two other fields of the same type as the variables x and y from tuple space; it also assigns the values in the tuple's second and third fields to x and y respectively:

C Version   **Fortran Version**
```
in("coord", ?x, ?y);                 in('coord', ?x, ?y)
```

If no matching tuple exists in tuple space, then the operation will block. The following tuple searches for the same sort of tuple, but specifies that the value in its second field must match the current value of x:

C Version   **Fortran Version**
```
in("coord", x, ?y);                  in('coord', x, ?y)
```

rd

The **rd** operation functions identically to **in** except that it does not remove the matching tuple from tuple space. For example, the following **rd** operation will attempt to match the same kind of tuple as in the examples with **in**, except that this time the value in the second field must be 3:

C Version	**Fortran Version**
`rd("coord", 3, ?y);`	`rd('coord', 3, ?y)`

When the **rd** operation successfully matches a tuple, the value in its third field will be assigned to the variable y. The tuple will remain in tuple space, available for use by other processes.

out

The **out** operation adds a tuple to tuple space. Prior to adding it, **out** evaluates all of its fields, resolving them to actual values. **out** returns after the tuple has been added to tuple space.

For example, the following **out** operation places a "coord" tuple into tuple space:

C Version	**Fortran Version**
`out("coord", 3, 10);`	`out('coord', 3, 10)`

If any of the fields in an **out** operation contain expressions, they are evaluated before the tuple is placed into tuple space. For example, this **out** operation will compute the value of the function f with the argument x, and then place a tuple with that value in its third field into tuple space (the second field will of course contain the current value of x):

C Version	**Fortran Version**
`out("coord", x, f(x));`	`out('coord', x, f(x))`

The evaluation order of the fields is not defined and cannot be relied upon. For example, in the following C-Linda **out** operation, x may or may not be incremented before it is used as the argument to the function f:

C Version	**Fortran Version**
`out("coord", x++, f(x));`	`out('coord', g(x), f(x))`

Similarly, in the Fortran-Linda version, there is no way to determine which routine will be called first should both f and g modify x.

These types of constructions should be avoided.

No specified action is taken if tuple space is full when an **out** operation attempts to place a tuple there. Current Linda implementations will abort execution and print a diagnostic message. Typically, such events are treated by programmers along the lines of a stack or heap overflow in conventional programming languages: the system is rebuilt with a larger tuple space. Future Linda versions may raise exception flags. Under TCP Linda, the size of tuple space is limited only by the total available virtual memory of all participating nodes.

eval

As we saw in the previous chapters, an **eval** operation creates a process tuple consisting of the fields specified as its argument and then returns. Here we'll go into more detail about how that happens.

Logically, the fields of an **eval** are evaluated concurrently by separate processes; **eval**ing a five-field tuple implicitly creates five new processes. When every field has been evaluated, then the resulting data tuple is placed into tuple space.

In current Linda implementations, however, only expressions consisting of a single function call are evaluated within the live tuple and actually result in a new process. These functions can use only simple data types as their arguments and return values (see below). All other fields are evaluated sequentially before new processes are created.

Here is a typical **eval** statement:

C Version

```
eval("coord", x, f(x));
```

Fortran Version

```
eval('coord', x, f(x))
```

This **eval** will ultimately result in the same data tuple as the **out** operation we looked at previously. However, in this case, the **eval** operation will return immediately, and a new process will evaluate `f(x)`. By contrast, the **out** operation will not complete until the evaluation of `f(x)` is complete and it has placed the data tuple into tuple space.

Compare these two C-Linda loops:

Loop with out

```
for (i=0; i < 100; i++)
  out("f_values", i, f(i));
```

Loop with eval

```
for (i=0; i < 100; i++)
  eval("f_values", i, f(i));
```

The loop on the left will sequentially evaluate `f(i)` for the first 100 non-negative integers, placing a tuple into tuple space as each one completes. The loop on the right will create 100 concurrent processes to evaluate the function `f` for each `i` value. As each process finishes, the resulting data tuple will go into tuple space.

eval's Inherited Environment

In C-Linda, the environment available to **eval**ed functions consists solely of the bindings for explicitly named parameters to the function. Static and global initialized variables in the function are not currently reinitialized for the second and following **eval**s and thus will have unpredictable values. The created process does not inherit the entire environment of its parent process. Thus, in the preceding **eval** example, the environment passed to the function `f` will include only the variable `x`.

Under Fortran-Linda, created processes inherit only the environment present when the Linda program was initiated,[†] with no modifications due to execution of user code. In many implementations, this is achieved by saving a clean copy of the program image from which to clone new processes.

[†] Some distributed memory Linda systems satisfy these semantics exactly only when the number of evals does not exceed the number of processors.

Consider the following program structure:

```
Block Data
Integer val
Common /params/val
Data val /5/
End

Subroutine real_main
Integer val
Common /params/ val

val = 1
Call f(3)
eval('worker', f(3))
...
Return
End


Subroutine F(I)
Integer val
Common /params/ val
...
Return
End
```

When subroutine `f` is invoked using the `Call` statement, the variable `val` will have the value `1`, since that value was assigned just prior to the call. However, when the subroutine is invoked using the **eval** statement, the variable will have the value `5`, since that was its initial value at the inception of the program.

eval Function Restrictions

evaled functions may have a maximum of 16 parameters, and both their parameters and return values must be of one of the following types:

- ◆ C-Linda: `int`, `long`, `short`, `char`, `float`, `double`
- ◆ Fortran-Linda: `integer`, `real`, `double precision`, `logical`

The first four C types can be optionally preceded by `unsigned`; the Fortran types may include a length specifier (e.g. `real*4`). Note that no arrays, structures, pointers, or unions are allowed as function arguments. Of course, data of these types can always be passed to a process through tuple space.

The other fields in an **eval** are also limited to the data types in the preceding list plus string constants.

Under Fortran-Linda, subprograms appearing in an **eval** statement can be either subroutines or functions. Subroutines are treated as if they were integer functions always returning the value 0. Functions must return values of type integer, logical, or double precision. Intrinsic functions may not be used in **eval** statements.

Predicate Operation Forms: inp and rdp

inp and **rdp** are predicate forms of **in** and **rd** respectively (that's what the **p** stands for). They attempt to match the template specified as their argument to a tuple in tuple space in the same way as **in** and **rd**, and they perform the same actual-to-formal assignment when a matching tuple is available. As expected, **inp** removes the matching tuple from tuple space, while **rdp** does not. When either of them successfully matches a tuple, it returns a value of 1 in C and .TRUE. in Fortran.

If no matching tuple is available, **inp** and **rdp** will not block. Rather, they will return a value of 0 (C) or .FALSE. (Fortran) and exit. Using **inp** and **rdp** can complicate your program, because they tend to introduce timing dependencies and non-deterministic behavior that may not have been intended.

For example, consider the following C-Linda code fragments:

```
/* Master code */
real_main()
{
    ...
    for (i=0; i<tasks; ++i) out("task", i);
       out("tasks outed");
    …
}

/* Worker code */
worker()
{
    ...
    rd("tasks outed");
    while (inp("task", ?i))
       do_task(i);
}
```

Clearly, if the **rd** of the "tasks outed" tuple were omitted, the the worker code would be non-deterministic. It might get any number of tasks before the loop terminated, which is not the intent. What is perhaps less clear is that the program is still non-deterministic *even with* the **rd**. This is due to the fact that out is asynchronous. There is no guarantee that all of the task tuples will be in tuplespace before the "tasks outed" tuple arrives.

It is a far better programming practice to use a counter or semaphore tuple in situations where **inp** or **rdp** seems called for. Consider this C-Linda example:

```
if (rdp("globals", ?x, ?y, ?z)==0)
   do_globals(x,y,z);
```

If the "globals" tuple is not available in tuple space, then **rdp** returns 0 and the process computes the globals itself. Simply doing a **rd** would result in blocking if the "globals" tuple weren't available, and recomputing them can be faster than waiting for them (although slower than reading them). The same effect can be accomplished via a tuple that can take on one of two values, for example:

```
("globals_ready", 0 or 1)
```

The master process **out**s this tuple with value 0 at the beginning of the program, and the process that computes the globals and sends that tuple to tuple space also **in**s this tuple and **out**s it again, changing its second field to 1. Then, the preceding code can be replaced by:

```
rd("globals_ready", ?i);
if (i)
    rd("globals", ?x, ?y, ?z);
else
    do_globals(x,y,z);
```

C-Linda Alternate Operation Names

The alternate names **__linda_in, __linda_rd, __linda_out, __linda_eval, __linda_inp,** and **__linda_rdp** are provided for cases where the shorter names conflict with other program symbols. Each alternate operation name begins with two underscore characters.

Specifying Tuples and Basic Tuple Matching Rules

This section discusses tuples and tuple matching rules in more detail and includes examples that use a variety of data types.

Tuples can have a maximum of 16 fields. In C-Linda, tuple fields can be of any of the following types:

- `int`, `long`, `short`, and `char`, optionally preceded by `unsigned`.
- `float` and `double`
- `struct`
- `union`
- Arrays of the above types of arbitrary dimension, including multidimensional arrays.
- Pointers must always be dereferenced in tuples.

In Fortran-Linda, tuple fields can be of these types:

- `Integer` (`*1` through `*8`), `Real`, `Double Precision`, `Logical` (`*1` through `*8`), `Character`, `Complex`, `Complex*16`
- Synonyms for these standards types (for example, `Real*8`).
- Arrays of these types of arbitrary dimension, including multidimensional arrays, and/or portions thereof.
- Named common blocks

Formal Tuple Matching Rules

A tuple and a template match when:

- ◆ They contain the same number of fields.

- ◆ All corresponding fields are of the same type.

  - • The type of a field containing an expression is whatever type the expression resolves to. The type of a field containing a formal is the type of the variable used in the formal.

  - • In C, for a structure or union field, the type is extended to include the structure or union tag name. The tag name and size of structures must match. Tagless unions and structures are not allowed.

  - • In Fortran, common blocks match based upon their name alone. Their internal structure is *not* considered.

  - • Arrays match other arrays whose elements are of the same type. Thus, an array of integers will match only other arrays of integers and not arrays of characters. Similarly, real arrays will not match integer arrays, even when they contain the same length in bytes.

  - • Scalar types don't match aggregate types. For example, if `a` is an array of integers, then `a:1` won't match an integer (but `a[2]` in C and `a(2)` in Fortran will). Similarly, in C, if `p` is a pointer to an integer, `*p` and `p:1` do not match (the `:n` array notation is discussed later in this chapter).

- ◆ The corresponding fields in the tuple contain the same values as the actuals in the template.

  - • Scalars must have exactly the same values. Care must be taken when using floating point values as actuals to avoid inequality due to round-off or truncation.

  - • Aggregate actuals such as arrays (which otherwise match) must agree in both the number of elements and the values of all corresponding elements.

The following sections contain many illustrations of these matching rules.

Scalars

The following C operations all place a tuple with an integer second field into the tuple space:

```
int i, *p, a[20], f();
p = &i;

out("integer", 3);/* constant integer */
out("integer", i);/* integer variable */
out("integer", *p);/* dereferenced ptr to int */
out("integer", a[5]);/* element of an int array */
out("integer", f(i));/* function returns an int */
out("integer", *++a);/* dereferenced ptr to int */
```

Note that single elements from an array are scalars of the type of the array.

The constructs `&i` and `p` are not included among these examples because each of them is a pointer to an integer; they are treated as arrays not as scalars (see the next section). Thus, the following tuple and template will not match even though `p` points to `i` (an integer):

```
("integer", p:1)
("integer", ?i)
```

Here are some example Fortran-Linda operations involving scalars:

```
        Real a(20), x
        Integer i
        Character *10 name

        out('integer', i)
        out('real', x)
        out('real', a(6))
        out('character', name)
```

Note that Fortran `Character` variables are scalars.

Arrays

Array handling within tuples and templates is very easy. Here are some examples of tuples and templates involving arrays:

C:
```
char a[20];
int len;

out("array1", a:);
in("array1", ?a:);

out("array2", a:10);
in("array2", ?a:len);
```

Fortran:
```
Dimension a(20)
Integer len

out('array1', a)
in('array1', ?a)

out('array2', a:10)
in('array2', ?a:len)
```

The general format for an array field in a tuple is `name:length`, where `name` is the array name, and `length` is the length of the array (number of elements). As the **out** operations in the examples indicate, the length is often optional. When it is omitted, the entire length of the array is assumed. In C, you must still include the colon when omitting the length, while in Fortran, the colon may also be omitted (although including it is correct too). For example, the first **out** operation in each language places the entire array `a` into the tuple space.

If you only want to place part of an array into the tuple space, then you can include an explicit length in the **out** operation. In this way, the second **out** operation in each language places only the first ten elements of array `a` into the "array2" tuple.

The array format is basically the same for arrays used as formals in templates. The one difference is that an integer variable is used for the `length` parameter, and its value is set to the length of the array in the matching tuple. Thus, the final pair of **in** operations in the preceding example will both result in the value of `len` being set to 10.

The semantics of Linda ensure that a user may alter the contents of an array immediately upon return from the **out**—the tuple generated by the **out** will still reflect the contents of the array just prior to the **out**. To enforce these semantics, Linda makes a copy of the array before returning from the **out**. Shared memory implementations of Linda allocate space for this copy in the shared memory region. Non-shared memory implementations of Linda allocate memory in the address space of the process executing the **out**.

Avoiding Array Copies

This copying can be problematic for very large arrays: It can nearly double memory requirements and add significantly to the time needed to process the tuple. Many non-shared memory Linda implementations provide a way for the user to request that the array not be copied. The user does this by negating the length. In this case, a supporting Linda implementation simply records the address of the data. This address is then used to locate the data that must be retrieved To satisfy a **rd** or an **in**. Any changes made to the data after the **out** but prior to retrieval will be reflected in the data sent. Since the user generally cannot know when the system internally retrieves the data, the tuple's content in the presence of such modifications has to be viewed as nondeterministic. While it is possible to design a parallel algorithm that exploits this non determinism, using a negated length field is tantamount, in most cases, to a promise (to oneself!) that the array will not be modified until, by some means, it is known that all **rd**s and **in**s against the tuple have completed. Refer to the release notes to see if a particular implementation supports this optimization. For more on arrays, Fixed Aggregates in C on page 36.

Pointers and Assumed-Size Arrays

C pointers to arrays and Fortran assumed-size arrays must *always* specify an explicit length when they are used as an actual within a tuple, as in these **out** operations:

C:
```
char b[20], *p, d[30];
int len;

p = b;
out("array2", p:20);
in("array2", ?d:len);
```

Fortran:
```
        Dimension b(*), d(30)
        Integer len

        out('array2', b:20)
        in('array2', ?d:len)
```

In both cases, the first twenty elements of array d are set to the value of the corresponding element in array b, and the variable len is set to 20. This requirement makes sense since these constructs can be used with arrays of different sizes within the same program.

The following **out** operations yield identical tuples:

C:
```
int *p, a[20];
p = a;

out("array", a:);
out("array", a:20);
out("array", p:20);
```

Fortran:
```
        Integer a(20)

        out('array', a)
        out('array', a:)
        out('array', a:20)
```

The length is required in the third C example; pointers must always specify an explicit length.

You could create a tuple containing the first ten elements of a with any of these operations:

C:
```
int *p, a[20], b[20], c[10], len;
p = a;

out("ten elements", a:10);
out("ten elements", p:10);
```

Fortran:

```
Integer a(20), b(*), c(10), len

out('ten elements', a:10)
```

and retrieve it with any of these operations:

C:

```
in("ten elements", ?a:len);
in("ten elements", ?p:len);
in("ten elements", ?b:);
in("ten elements", ?c:);
```

Fortran:

```
in('ten elements', ?a:len)
in('ten elements', ?a:)
```

All of the operations will retrieve ten integers, and `len` will be assigned the value 10 for those operations including it. Note that omitting the length variable is allowed, and such statements still will retrieve whatever number of elements is present in the "ten elements" tuple.

Assuming that the first ten elements of array b have the same values as the corresponding elements of array a, the following **in** operations would consume one of the "ten element" tuples without assigning any values:

C:

```
in("ten elements", b:10);
```

Fortran:

```
in('ten elements', b:10)
```

Here array b is used as an actual, and a matching tuple is simply removed from the tuple space (assuming one exists). Note that there are better ways to remove tuples from the tuple space without copying the data in them, as we will discuss later.

Array references can begin with any desired element, as in these examples:

C:

```
in("ten elements", ? &b[4]:len);
```

Fortran:

```
in('ten elements', ?b(5):len)
```

These operations would retrieve a "ten elements" tuple, place the ten array elements it held into the fifth through fourteenth elements of array b, and set the value of `len` to 10. Although the examples in this section have used integer arrays, exactly the same principles and syntax apply when accessing arrays with elements of other types.

Note that retrieving a tuple containing an array longer than the target array used in the template will result in writing past the end of the target array.

Multidimensional Arrays

In Fortran, array shape is ignored, and so multidimensional arrays can be handled in a similar way to one-dimensional arrays. If you want to refer to a subsection of a multidimensional array, you can do so by specifying a starting element and/or length, but a more powerful mechanism for doing so is provided by the Fortran 90 array syntax, described in the next section.

The remainder of this section will discuss multidimensional arrays in C.

In C, the basic principle to keep in mind is that multidimensional arrays match only when their types and shapes are the same; the same holds true for sections of multidimensional arrays. We will use these arrays in the examples:

```
int a[100][6][2], b[60][4][5], d[100][6][2];
```

The following operations create and retrieve a tuple containing a multidimensional array:

```
out("multi", a:);
in("multi", ?d:);
```

The following **in** operation will not succeed because the shapes of arrays a and b are different (even though they have the same number of elements):

```
out("multi section", a:);
in("multi section", ?b:);/* WILL NOT WORK */
```

Portions of multidimensional arrays may also be specified. Here are some examples:

```
int a[3][5][2], b[5][2], c[2], i;

out("section", a[0][0]:);
in("section", ?c:);

out("2d section", a[0]:);
in("2d section", ?b:);

out("not an array", a[0][0][0]);
in("not an array", ?i);    /* just a scalar ... */
```

In the first pair of operations, the construct `a[0][0]:` points to the start of an array of length 2 which is why the formal involving array c matches it. In the second pair of operations, two 5x2 array sections (with the second one consisting of the entire array b) will match.

The last example is a bit tricky; in this case, the **out** operation creates a tuple with the first element of a as its second field, and any integer can be used in a formal to retrieve it.

Fortran 90 Array Sections

Fortran-Linda recognizes a subset of the array syntax used in the Fortran 90 standard within its operations. This syntax provides an alternate way of referring to arrays and their subsections and may not be combined with the *name:length* notation we've considered so far.

Array subscript references in Fortran-Linda may take the following form:

[*ifirst*]:[*ilast*][:*istride*]

where *ifirst* is the smallest index of a slice of the array, *ilast* is the largest index in the slice, and *istride* is the stride (if omitted, the stride defaults to 1). A full array section is specified with this type of expression for each dimension in the array. The shorthand form of a single colon alone refers to the entire range of values for that array dimension with stride 1.

Here are some examples:

```
real a(100,100,100), b(100,100)

out('whole array', a(1:100,:,1:100))
out('second row', b(2:2,:))
out('divide in 2--part 1', a(:,:,:50)
out('divide in 2--part 2', a(:,:,51:)
out('every other', b(1:100:2,1:100)
```

The first **out** operation places the entire array a into the tuple space. The second places only the second row of array b. The third and fourth operations divide array a in half and place the two halves into the tuple space, defaulting the beginning and ending elements in the third dimension to the first and last array element, respectively. The final operation illustrates the use of the stride value.

Fortran Named Common Blocks

Entire named common blocks can be transferred to a tuple space as a single unit. Named common blocks are referenced by their common block name, enclosed in slashes. For example, the following operations place and retrieve the designated common block:

```
Common /example/a,b,c,d,n

out('common', /example/)
in('common', ?/example/)
```

The **out** operation places the entire common block into the tuple space, and the following **in** operation matches and retrieves this same tuple. For matching purposes, common blocks with identical names match, and the internal structure of the common block is ignored. Blank common may not be placed into tuples using this method. The best solution in such cases is usually to convert it to a named common block.

C Structures

From the point of view of syntax, structures work very much like scalars. For example, these two **out** operations create identical tuples:

```
struct STYPE s, t, *p;

p = &s;
out("structure", s);
out("structure", *p);
```

Either of these **in** operations will retrieve one of the tuples created by the previous **out** operations:

```
in("structure", ?t);
in("structure", ?*p);
```

Structure fields are one way to create tuples containing one record of data. For example, the following loop retrieves NREC records from a database and places them into the tuple space. Each record is identified by the integer record number in the tuple's second field:

```
int i;
struct REC s;

for (i=0; i < NREC; i++) {
   get_next_rec(&s);
   out("record", i, s);
   }
```

Structures may also be used as actuals in templates:

```
in("structure", t);
```

In this case, the structures in the tuple and template must have the same structure tag, the same size, and they must be identical on a byte-by-byte basis, including values in any padding bytes. When using such constructs, be careful to take structure padding into account.

An array of structures is treated just like any other array:

```
int len;
struct STYPE s[20], t[20];

out("struct array", s:10);
/* matches; sets len = 10 */
in("struct array", ?t:len);
```

Varying-Length Structures

You can use the colon notation with structures to specify them as varying. In this case, the length is taken to be the size of the structure in bytes. This construct was designed for structures having varying arrays as their last elements.

Here are two examples:

```
struct STYPE {
   double  a,b,c;
   int     buf_len;
   int     buf[1]
};
int bytes;
STYPE *s;

/* declared struct length */
out("varying struct", *s:);

bytes = sizeof(STYPE) + (sizeof(int) * (buf_len-1));
/* current structure length */
out("varying struct", *s:bytes);
```

The first **out** operation creates a tuple with a varying structure as its second field. The second **out** operation creates a tuple whose second field is also a varying structure; for this instance, the current length is set to the size in bytes of STYPE (including the first element of array buf) plus the size in bytes of the remainder of array buf (the product of the size of one element and its number of elements minus 1).

C Character Strings

In keeping with C usage, character strings are simply arrays of characters, and the normal array considerations and matching rules apply. The only exception occurs when comparing string constants with character arrays.

The length of a string constant is the number of characters in it plus one, for the null terminator. Thus, the string "hello" has a length of 6, and a five-element character array will not match it; it requires a six element array:

```
char s[6];
int len;

out("string", "hello");/* length = 6 */
in("string", ?s);
```

The array colon notation may also be used with strings for fields where the length of the string is variable:

```
out("string", "hi":);
in("string", ?s:len);
```

Note that the literal string in the **out** operation needed to include the colon in order for the template in the **in** operation to match since the template does not require a string of a specific length. This requirement holds for such templates even when the length of the literal string in the tuple and the declared length of the array used in the template are the same—even if `hi` had been `hello`, the colon would still be needed in the **out** operation.

Anonymous Formals

A formal need not include a variable name, but instead can use a data type name in C, and in Fortran can use a `typeof` construct with a data type name as its argument. Here are some examples:

C:
```
struct STYPE s;

in("data", ?int);
in("struct", ?struct STYPE);
in("int array", ?int *:);
```

Fortran:
```
Common /sample/array,num,junk

in('data', ?typeof(integer))
in('array', ?typeof(real*8):)
in('common', ?typeof(/sample/))
```

Such formals are called anonymous formals. Anonymous formals within **in** operations still remove a tuple from the tuple space, but the data in that field is not copied to any local variable. It is effectively thrown away. This construct can be useful when you want to remove a tuple containing, for example, a large array from a tuple space. Anonymous formals allow it to be removed without the time-consuming copying that would result from an **in** operation.

A **rd** operation with a template containing only actuals and anonymous formals has no effect if there is a matching tuple, but still blocks when none is available.

Fixed Aggregates in C

In C, you can reference fixed length aggregates such as arrays in tuples simply by name (i.e., without the colon notation). For example, if `a` is an array of declared length 20, then it can be referred to in a tuple by its name alone, as in this example:

```
int a[20], b[20];

out('array', a)
```

Similarly, you can retrieve such an array by name:

```
rd('array', ?b)
in('array', ?a)
```

You can also treat multidimensional arrays (and sections thereof) this way:

```
int a[100][6][2], b[25][24][2];

out("multi section", a[0][0]);
in("multi section", ?b[0][0]);
```

In these statements, both arrays are essentially being used as pointers to the start of a fixed aggregate of length 2. Two array sections treated as fixed aggregates in this way must have the same length and shape to match.

Fixed aggregates *never* match arrays specified in the usual way. Fixed aggregates match only other fixed aggregates. Literal character strings specified without a terminal colon are treated as fixed aggregates. Thus, neither of the following **in** operations will find a match:

```
char a[20], s[6];
int len;

out("wrong", a:);
out("also wrong", "hello");

in("wrong", ?a);/* no match */
in("also wrong", ?s:len);/* no match */
```

Termination of Linda Programs

Since a Linda program can consists of many concurrent processes, program termination requires a bit more care than for sequential programs. Linda programs can terminate in one of three ways:

- ◆ The program can terminate normally when the last process (usually `real_main`) finishes (returns from its outermost call). A program may terminate by having the final process call the C-Linda support function `lexit` (`flexit` is the Fortran-Linda form), but this is not required. Note that if you want to call an exit function, use `lexit`; *never* use the standard system call `exit`.

- ◆ The program can force termination by calling `lhalt` (C) or `flhalt` (Fortran). When such a call occurs in any process of the Linda program, the entire computation will be terminated immediately and automatically.

- ◆ If any process terminates abnormally, then the entire program will again be terminated automatically at once.

An individual process within the parallel program may terminate execution of user code by calling `lexit` or `flexit`. This function terminates local user processing without affecting the other running processes (if any).

See the <u>Linda Usage and Syntax Summary</u> for full details on available termination routines and their use.

Example: Freewake

We're now ready to look at another Linda program which illustrates all four operations and several tuple matching principles in action. This program also illustrates the following techniques:

- ◆ The master process becoming a worker after initialization is complete.
- ◆ The use of a composite index to combine two existing indices.

The program we'll examine is named **Freewake**, a computational fluid dynamics application. It was developed to model the wake structure created by helicopter rotor blades and its influence on the blades themselves to aid in the design of new helicopters. It was originally parallelized with C-Linda, but we will provide both C-Linda and Fortran-Linda versions here.

At its core, the calculation is essentially a very complex *N*-body problem: the wake surface is divided into a large number of discrete elements. The structure of this surface depends on the properties of the individual elements, including their velocities. For each time step, the change in velocity of each element is a function of its interactions with all of the other elements, and all of these changes in velocity determine the new structure of the wake surface. Once it is obtained, the program calculates the interaction of the wake and the blades themselves.

The vast majority of the computation is spent calculating the displacement of each point of the wake for each time step. This is computed by these three nested loops. (See <u>How and Where to Parallelize</u> on page 14 for information about determining the most computationally intensive portions of a program.) Here is the original Fortran code:

```
DO I = 1, NBLADES! Typically = 2
DO J = 1, NFILMNT! Typically = 16
DO K = 1, NSEGMNT! Typically = 512
   CALL CALCDISP( X(I,J,K), Y(I,J,K), Z(I,J,K),
          DX(I,J,K),DY(I,J,K),DZ(I,J,K))
END DO; END DO; END DO
```

The subroutine `CALCDISP` calculates the displacement of a single point in three dimensional space. Each call to `CALCDISP` is independent. It takes the x, y, and z-coordinates of a point in space (elements of the arrays `X`, `Y`, and `Z`, respectively), and produces the displacement in each direction of the specified point in space as its only output (elements of the arrays `DX`, `DY`, and `DZ`). Thus, performing some of those calls at the same time would have a large effect on overall program performance.

Here is the key part of the code which serves as the master process and coordinates the calls to CALCDISP:

C Version

```c
/* put data into tuple space */
out ("wake", x, y, z, nblades, nfilmnt, nsegmnt);

out ("index", 0)                    /* create task counter */
for (i = 0; i < NWORKERS; i++)   /* start workers */
   eval("worker",worker());
worker();                           /* then become a worker */

for (index = 0; index <= nblades * nfilmnt; index++) {
   /* gather data from tuple space */
   in("delta", index, ?tmp_x, ?tmp_y, ?tmp_z);
   Put data into the displacement arrays DX, DY, and DZ.
   }
```

Fortran Version

```fortran
C     Put data into tuple space
      out('wake', x, y, z, nblades, nfilmnt, nsegmnt)

C     Create task counter, start workers, then become
C     a worker yourself.
      out('index', 0)
      Do 10 I=1,NWORKERS
         eval('worker', worker())
 10   Continue
      Call worker

      Do 20 index=0,nblades*nfilmnt-1
         in('delta', index, ?tmp_x, ?tmp_y, ?tmp_z)
         Put data into the displacement arrays DX, DY, and DZ.
 20   Continue
```

The first **out** operation places the position arrays x, y, and z into tuple space in the "wake" tuple; later, the workers will each **rd** it. Then, the master process creates the "index" tuple, from which the workers will generate tasks. In the first **for** loop, the master process next starts NWORKERS worker processes.

At this point, the master process has completed all necessary setup work, so it becomes a worker itself by calling the same worker function used in the **eval** operations. This is a common technique when few startup activities are required, and worker processes run for a long time. If the master did not become a worker, then its process would remain idle until the workers finished.

After the workers finish, the master process executes the final **for** loop, which gathers the results produced by all the workers, removing them from tuple space and placing them in the locations expected by the original Fortran program.

Here is a simplified version of the worker routine:

```
worker()
{
   rd("wake", ?x, ?y, ?z, ?nblades, ?nfilmnt, ?nsegmnt);

   while (1) {                    /* loop until work is done */
      in ("index", ?index);   /* get current task index */
      out("index", index+1);  /* increment and put back */
      if (index >= nblades * nfilmnt)  /* test if done */
         break;

      iblade = index / nfilmnt;          /* blade */
      ifil = index % nfilmnt;            /* filament */
      for (iseg=0; iseg<nsegmnt; iseg++)  /* segment */
         calcdisp_(&x[iseg][ifil][iblade],
            &y[iseg][ifil][iblade], &z[iseg][ifil][iblade],
            &dx[iseg], &dy[iseg], &dz[iseg], &x, &y, &z);
      /* place results in tuple space */
      out("delta", index, dx, dy, dz);
   }
}
```

Fortran Version

```
      Subroutine worker
      Double Precision x(*), y(*), z(*)

      rd('wake', ?x, ?y, ?z, ?nblades, ?nfilmnt, ?nsegmnt)

      Do 10 I=1,VERY_BIG_NUM
         in('index', ?index)
         out('index', index+1)
         if (index .GE. nblades*nfilmnt) Return

         iblade=(index / nfilmnt) + 1
         ifil=modulo(index,nfilmnt)+1
         Do 20 iseg=1,nsegmnt
            call calcdisp(x(iblade,ifil,iseg),
     *         y(iblade,ifil,iseg), z(iblade,ifil,iseg),
     *         dx(iblade,ifil,iseg), dy(iblade,ifil,iseg),
     *         dz(iblade,ifil,iseg))
 20      Continue

         out('delta', index, dx, dy, dz)
 10   Continue
      Return
      End
```

Each worker process first reads the position arrays and their index limits from tuple space. The worker then loops continuously until all points are done. At the beginning of each loop, it removes the "index" tuple from tuple space, increments the counter in its second field, and then returns it to tuple space for possible use by other processes.

This counter, stored in the variable `index`, serves as a composite index, combining the outer two loops of the original Fortran code. Each task consists of executing the inner Fortran loop for a fixed pair of `I` and `J` values (i.e., specific blade and filament indices). A single counter tuple is easily retrieved, incremented, and returned to tuple space.

There are `NFILMNT` times `NBLADES` distinct pairs of `I` and `J` values, so the worker first tests whether the counter's value is greater than or equal to this product; equality is included in the test since `index` begins at 0 and runs to `(nfilmnt*nblades)-1` (we've given the equivalent variables in the Linda programs lowercase names). The variable `iblade` is defined as `index` divided by `nfilmnt`, and `ifil` is defined as `index` modulo `nfilmnt`. Because these are integer operations and all fractions are truncated, `iblade` will remain 0 until `index` reaches `nfilmnt`, then become and remain 1 until `index` reaches `nfilmnt*2`, and so on. At the same time, `ifil` counts from 0 to `nfilmnt-1` for each value of `iblade` over the same period. The following table indicates the encoding of `iblade` and `ifil` within `index` for `nfilmnt=3` and `nblades=2`.

	blade	*filament*
`index`	`iblade`	`iseg`
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2
6	*terminate*	

Once `iblade` and `ifil` are computed, a **for** loop calls a slightly modified version of the original `calcdisp` routine once for each segment value, using the computed `iblade` and `ifil` values each time. In the C-Linda version, this loop differs from the Fortran inner loop it replaces in several ways. First, the arguments to `calcdisp` are explicitly the addresses of the relevant array elements, since Fortran subroutine arguments are always passed by reference. Second, the first and third array indices are reversed, due to the differing Fortran and C multidimensional array ordering conventions. The Fortran arrays have not changed in any way, so the location denoted by the Fortran `X(i1,i2,i3)`, for example, must be accessed as `x[i3][i2][i1]` from C. Third, C arrays begin at 0 while the Fortran arrays begin at 1. However, this is easily accounted for by making `iseg` run from 0 to `nsegmnt-1` rather than from 1 to `nsegmnt` (as it did in the Fortran loop). Fourth, we've also added the addresses of the arrays `x`, `y`, and `z` to the subroutine's arguments (in Fortran, `CALCDISP` accesses them via a COMMON block not shown here). Finally, we've translated the subroutine name to lowercase and appended an underscore, a common requirement when calling a Fortran subroutine from C.

After the inner loop completes, the worker sends the displacement values it has created to tuple space, and the outer loop begins again. When the counter in the "index" tuple exceeds its maximum value, each worker process will terminate the next time it examines it.

As we've seen, it was very easy to transform **Freewake** into a parallel program with Linda because all of its work is isolated so well within a single routine. In the next chapter we'll look at several more complicated case histories.

3
Using TCP Linda

This chapter describes the special considerations involved when executing Linda programs on a network of computers.[†] The network version of Linda is called TCP Linda, and that terminology is used here as well. Release notes for particular platforms often contain additional information relevant to the use of TCP Linda.

Quick Start

Running parallel programs on networks is complicated by issues such as process scheduling, executable location, remote execution facilities, and the like. **ntsnet** is a powerful, flexible utility for executing Linda programs on a network, designed to help you manage this complexity. This section discusses the simplest possible case, and is designed to enable you to get started running programs right away. The remainder of the chapter covers more complex scenarios and the **ntsnet** features provided to handle them.

Normally, TCP Linda runs the `real_main` process on the local system, and **eval**ed worker processes run on remote hosts. This requires that it is possible to successfully execute an **rsh** command from the local host to each remote host without being required to enter a password. Consult the man page for **rsh** or your system administrator if this condition does not hold true for your site.

In the simplest case, the current working directory is a commonly-mounted directory, accessible by the same pathname from every host that you want to use (it can be a permanent mount point or be auto-mounted).

Given these assumptions, the following steps are necessary to create and run a TCP Linda program:

* Make sure that the `bin` subdirectory of the Linda distribution tree is in the search path.

* Define the set of hosts to be used for program execution by creating the file `.tsnet.config` in your home directory, containing a single line like this one:

    `Tsnet.Appl.nodelist: moliere sappho blake shelley`

    Replace the sample node names with the appropriate ones for your network.

[†] TCP Linda is also available for certain distributed memory parallel computers. However, the discussion here centers on networks, although identical considerations apply in both cases. This discussion applies only to TCP Linda version 2.4.7 or higher.

- Compile the program, using a command like the following:

```
$ clc -o hello_world hello_world.cl
```

See the "Quick Start" section of Chapter 2 for a more detailed discussion of this and the following step.

- Finally, run the program, preceding the normal invocation command and arguments with **ntsnet**:

```
$ ntsnet hello_world 4
```

ntsnet will automatically run the program on the defined set of nodes.

What ntsnet Does

ntsnet is responsible for the following tasks:

- Parsing the command line options and configuration file entries.
- Querying remote systems for load averages.
- Locating local executable files.
- Determining what set of nodes to run on, based on its scheduling algorithm.
- Determining working directories and executable file locations on remote nodes, using map translation and the associated configuration files (if necessary).
- Copying executable files to remote nodes (if necessary).
- Initiating remote processes.
- Waiting for normal or abnormal termination conditions during program execution.
- Shutting down all remote processes at program termination.
- Removing executables from remote systems (if appropriate).

The remainder of this chapter will look at these activities—and the ways that the user can affect how **ntsnet** performs them—in considerable detail.

Using the ntsnet Command

The general syntax for the **ntsnet** command is:

ntsnet [*options*] *executable* [*arguments*]

where *options* are **ntsnet**'s options, *executable* is the executable file to run on the local system, and *arguments* are command line arguments for the specified network program. **ntsnet** uses the command line options, the location of the local executable, and the settings in its configuration files to determine all of the remaining information it needs to execute the network parallel program.

Customizing Network Execution

TCP Linda provides the **ntsnet** command to execute parallel programs across networks of machines. **ntsnet** is designed for maximum flexibility. Proper configuration makes running a TCP Linda program as simple as prepending its executable's pathname—and any arguments—with the **ntsnet** command, as in the previous example.

ntsnet can draw its configuration information from a variety of sources. These sources are, in order of precedence:

- ◆ command line options
- ◆ **ntsnet**'s application-specific configuration file (if any)
- ◆ **ntsnet**'s local configuration file
- ◆ **ntsnet**'s global (system-wide) configuration file
- ◆ **ntsnet**'s built-in default values

When they do not conflict, the settings from all of these sources are merged together to create the **ntsnet** execution environment.

We'll cover each of these items separately, in the context of actual execution tasks. See the <u>Linda Usage and Syntax Summary</u> for a complete reference to all command line options and configuration file resources and formats.

ntsnet Configuration Files

ntsnet uses several configuration files: the global, local, and application-specific *configuration files*—we'll use this term to refer to the specific files rather than in a generic sense from this point on—which define program execution characteristics, and the local and global *map translation files*, which define directory equivalences on the various potential execution nodes in the network.

ntsnet first looks for an application-specific configuration file, named `tsnet.config-`*application_name*, where *application_name* is the name of the application being executed with **ntsnet** (application names will be discussed shortly). ntsnet looks for an application-specific configuration file in the following way: first, if the executable on

the command line contained a full or partial directory specification, that location is searched for this configuration file. If only an executable filename was given, then the directories in the `TSNET_PATH` environment variable are searched in turn.

The optional local configuration file and map translation file are named `.tsnet.config` and `.tsnet.map` respectively. If used, these must be located in the user's home directory. The global files—`tsnet.config` and `tsnet.map`—are located in the `common/lib` subdirectory of the Linda tree: for example, in `/usr/sca/linda7.1/common/lib` if the Linda tree begins at `/usr/sca/linda7.1`. Settings in the local files always take precedence over those in the global files, and settings in the application-specific file take precedence over the local file. The global files may also be ignored entirely if desired. Note that precedence operates on a setting-by-setting basis, and not by an entire file.

The **ntsnet** configuration files contain entries which specify various execution parameters and desired characteristics. Depending on the parameter to which they apply, entries can vary significantly in their ultimate scope; they can affect:

- The execution of any TCP Linda program
- The execution of a specific program on every node it uses
- The execution of any program on a specific node
- The execution of a specific program only on a specific node

The configuration file syntax is modeled after the Xlib resources of the X Window System. However, **ntsnet** and TCP Linda are neither part of X nor do they require it. Some users may find a general introduction to X resources helpful; see the <u>Bibliography</u> for the appropriate references.

A resource is basically just an execution parameter (characteristic). The configuration file defines values for the various available resources and specifies the contexts—application programs and/or execution nodes—for which the value applies.

The **ntsnet** configuration files consist of lines of the following form:

> *program*[`.`*application*][`.`*node*]`.`*resource*`:` *value*

where *program* is the system program name to which the resource being defined is applied, *application* is the relevant user application program, *node* is the relevant node name, *resource* is the resource/characteristic name, and *value* is the value to be set for this resource in this context. We'll look at each of these parts in turn.

To begin with, the system program for the **ntsnet** configuration file entries will always refer to TCP Linda. We'll look at the exact syntax of this component in a moment. At this point in time, this component is simply a carryover from the X syntax.

A resource can be either *application-specific*, *node-specific*, or apply to both applications and nodes. For example, the resource **rworkdir**, which specifies the working directory on a remote node, is application and node specific, meaning that a different value can be set for it for every application program and node combination. For example, you can specify

a different working directory when running program `bigjob` on node *moliere* and on node *chaucer*, and you can specify different working directories for programs `bigjob` and `medjob` on node *moliere*.

In contrast, the resource **speedfactor** is node-specific, meaning that you can specify a different value for each potential execution node, but not for node-application program combinations; the value for a node applies to all TCP Linda applications that run on it. Such resources usually specify intrinsic characteristics of a node which don't depend on the application program being run. For example, **speedfactor** specifies how fast a node is relative to other nodes in the network, something which is relatively constant across different applications (at least in theory).

Finally, the resource **maxprocspernode** is an application-specific resource, meaning that its value can be specified separately for different application programs. The value set for an individual application is used for whatever nodes it may execute on. This resource specifies the maximum number of processes per node that can be run for a given application program (the default is 1).

Here are some example entries:

```
! some sample .tsnet.config file entries

ntsnet.hello_world.moliere.rworkdir: /tmp

ntsnet.hello_world.maxprocspernode: 1

ntsnet.moliere.speedfactor: 2
```

Lines beginning with an exclamation point (!) are comments. The second line sets the working directory to `/tmp` on the node *moliere* when the application `hello_world` is run there. The third line sets the maximum number of processes that can run on any one node when the application `hello_world` is executing to 1, and the final line sets the speed factor for the node *moliere* to 2—where the default value is 1—indicating that it's about twice as fast as the norm.

In configuration file entries, the program, application and node components (if used) can be either the class name or a specific instance of a class. Class names—recognizable by their initial capital letter—act as wildcards, stating that the entry applies to all instances of the specified class. In this way, they can serve as default values for specific instances— specific applications and/or nodes—for which no explicit entries have been created. The following table lists the possible values for each of these three components of a configuration file entry:

Item	Associated Class Name	Example Instance
program	`Tsnet`	`ntsnet`
application	`Appl`	*user application program name*
node	`Node`	*node name, user-defined node resource*

Currently, `ntsnet` is the only valid specific instance of the class `Tsnet`, so the two are effectively equivalent. Thus, for every entry, the program component will be either the class `Tsnet` or its only specific instance, `ntsnet`.

Application names are usually the name of the corresponding executable program. In order to make the separations between components possible, however, periods in application names must be translated to underscores. Thus, the application `big.job` would appear in configuration file entries as `big_job`. Application names can also be user defined if desired, and the application name to use for a given run can be specified on the command line with **ntsnet**'s **-appl** option. For example:

`$` **`ntsnet -appl big_job medium.job`**

This option can be used either to apply one application's settings to a different application program or to specify the use of a user-defined application name (which need not correspond to the name of any executable program). Note that periods in executable names are translated to underscores only when used as application names in the configuration files; such translation should not take place at any other time, such as when they are invoked in the **ntsnet** command line.

Node names can be full node names, such as *moliere.frachem.com*, or node nicknames: *moliere*. In the node component of configuration file entries only, periods again have to be translated to underscores (so that **ntsnet** can figure out where the component boundaries are). Anywhere else in the configuration file—as part of resource values, for example— and on the command line, no such translation is used.

Configuration file resources are keywords whose values specify some kind of execution behavior option or application or node characteristic. Consider these examples:

`Tsnet.Appl.Node.rworkdir: /tmp`

`Tsnet.Appl.maxprocspernode: 4`

`Tsnet.Node.speedfactor: 1`

These three entries all refer to application and node classes only, thereby serving as default values for instances not specifically defined in other entries. The first line sets the default working directory for remote nodes to `/tmp` on each node. The second line sets the maximum number of processes per node to 4, and the final line sets the default node speed factor to 1. These entries are completely general, applying to every application program and/or node. Contrast them to the earlier example, which applied only to the explicitly named applications and nodes.

One can also freely intermix classes and specific instances, as in these examples:

```
Tsnet.hello_world.Node.rworkdir: /tmp/hello
Tsnet.Appl.moliere.rworkdir: /xtmp
```

The first example sets the default working directory to `/tmp/hello` for any remote node when the program `hello_world` is run. The second example sets the default working directory on node *moliere* to `/xtmp` whenever a network application without its own entry for this resource runs remotely on it.

Many resources have corresponding **ntsnet** command line options. These options are designed to be used to override configuration file settings for a specific program run (although they can be used instead of the configuration files if desired). Here is an example command which runs `hello_world`, overriding its usual **maxprocspernode** resource setting:

```
$ ntsnet -maxprocspernode 3 hello_world ...
```

Command line options override any configuration file settings. Remember also that local configuration file settings take precedence over those in the global (system-wide) configuration file.

Resource Types

In addition to their scope—node-specific, application-specific, or node and application specific—resources can also be classified by the kind of value they expect.

Some resources, like those we've looked at so far, require a value: an integer or a directory, for example. Many others are Booleans, and expect a true or false setting for their value. For such resources, the following values are all interpreted as true: true, yes, on, 1. These values are all interpreted as false: false, no, off, 0. For example, the following entry indicates that the node *marlowe* is not currently available:

```
Tsnet.marlowe.available: no
```

ntsnet command line options corresponding to resources taking Boolean values use the following syntax convention. If the option name is preceded by a minus sign, then the resource is set to true, and if it is preceded by a plus sign, the resource is set to false. For example, the command line option **-useglobalconfig** sets the resource **useglobalconfig** to true, stating that the global **ntsnet** configuration file should be consulted. The option **+useglobalconfig** sets the resource to false, and the global configuration file will be ignored. The polarities of the plus and minus signs may seem counterintuitive at times; just remember that minus means on, which is the usual convention used by most X applications.

Note that all options which require parameters—for example, the command line option **-maxprocspernode** option we looked at earlier—are preceded by a hyphen (a prepended plus sign has no meaning for them and will generate an error). Their values follow them, separated by a space.

Not all resources have named command line options. The **-opt** option is provided so that any resource's value can be specified from the command line. It takes a valid configuration file entry as its parameter, enclosed in double quotation marks:

```
$ ntsnet -opt "Tsnet.moliere.available: no" hello_world …
```

A third type of resource enables users to create named node lists. Here are some examples:

```
Tsnet.Appl.fran: moliere gaughin voltaire pascal

Tsnet.Appl.eng: chaucer marlowe blake joyce

Tsnet.Appl.chem: @sparcs @rs6k priestley dalton
```

Each of these lines defines a name for a list of nodes. The first line defines a list of nodes to be associated with the name `sparcs`, for example. When a list name is used as a component in another list, its name is preceded by an at sign (to indicate resource indirection), as in the third line above. Up to 16 levels of indirection are allowed.

We've now introduced all the pieces of the **ntsnet** configuration file. The following sections will introduce many of the specific resources in the context of TCP Linda execution scenarios.

Determining Which Nodes a Program Will Run On

Two resources control what nodes a given application will run on. First, the **nodelist** resource, which takes a list of nodes as its value, specifies a node list for a given application. Here are some examples:

```
Tsnet.Appl.nodelist: @chem gauss newton descartes

Tsnet.hello_world.nodelist: gauss moliere dalton avogadro
```

The first line specifies the default set of execution nodes for TCP Linda programs (in addition to the local node). The second line specifies a different set for the application `hello_world` (which overrides the default value set in the first line).

Duplicates are automatically removed from node lists. Variant name forms for the same node—the full name and the nickname, for example—are also discarded (the first one is kept). In such cases, a warning message is printed.

The **nodelist** resource can also take the special value `@nodefile`. This indicates that the contents of the file specified in the **nodefile** resource contains the list of nodes to be used (one name per line). If **nodefile** has not been given a value, then the file `tsnet.nodes` in the current directory is used. The value for **nodelist** defaults to `@nodefile`, so ignoring both of these resources will result in the same behavior as under previous releases of Network Linda (which used the **tsnet** command), providing backward compatibility if you do nothing.

The **-nodelist** and **-nodefile** command line options correspond to the **nodelist** and **nodefile** resources respectively. The value for **-nodelist** must be enclosed in double quotation marks if multiple nodes are listed:

```
$ ntsnet -nodelist "moliere leopardi sappho" hello_world 4
```

Specifying Execution Priority

Two resources control the execution priority of processes started by **ntsnet**. The **high** resource (application-specific) indicates whether processes are nice'd or not (it defaults to true). If **high** is set to false, then processes are run at lowered priority. The command line options **-high/+high** (abbreviable to **-h** and **+h**) correspond to this resource.

The **nice** resource (node and application specific) allows you to specify whether processes should be nice'd or not on a node-by-node basis for each application. For example, the following lines state that processes running `hello_world` on *moliere* should be nice'd but those on *chaucer* shouldn't be (although generally processes that run on *chaucer* are nice'd):

```
Tsnet.hello_world.moliere.nice: true

Tsnet.hello_world.chaucer.nice: false

Tsnet.Appl.chaucer.nice: true
```

The default value for the **nice** resource is true. If the value of the **high** resource is set to true, then it overrides the setting of the **nice** resource.

How ntsnet Finds Executables

ntsnet locates the local executable—the executable file that will run on the local node (which is the node where the **ntsnet** command is executed)—in the following manner. If a full pathname is specified on the command line, that path specifies the exact location of the local executable. For example, the following command executes the network program `hello_world` in `/tmp`:

```
$ ntsnet /tmp/hello_world …
```

If only an executable name is specified, then **ntsnet** uses the `TSNET_PATH` environment variable to locate the executable file. The environment variable's value should be a colon separated list of directories, which are searched in order for required executables (working just like the UNIX `PATH` variable). If `TSNET_PATH` is unset, it defaults to:

```
/usr/bin/linda:.
```

This means that first the directory `/usr/bin/linda` is searched, followed by the current directory.

The location of the local executable can play a large part in determining the locations of the executable files to be run on remote nodes using **ntsnet**'s map translation feature (described below). These remote directory locations are also explicitly specifiable using the **rexecdir** resource (application and node specific). Here is an example:

```
Tsnet.Appl.Node.rexecdir: /usr/local/bin

Tsnet.Appl.moliere.rexecdir: /usr/linda/bin

Tsnet.hello_world.Node.rexecdir: /usr/bin

Tsnet.hello_world.moliere.rexecdir: /usr/linda/bin/test
```

The first line sets the default location for remote executables to `/usr/local/bin` on the remote system, meaning that **ntsnet** should use this directory on each node as the default location for the executable file to startup. Subsequent lines set different default values for the application `hello_world` and for the node *moliere*. The final line sets a specific value for the application `hello_worl`d when running remotely on *moliere*: the executable for `hello_world` on *moliere* resides in the directory `/usr/linda/bin/test`.

The **rexecdir** resource can also be given the special value `parallel` (which is its default value). This indicates that map translation is to be performed on the directory where the local executable resides, for every node which has no specific remote execution directory set. Map translation involves taking the name of the local directory containing the executable to be run and determining what the equivalent directory is for each remote node participating in the job. These equivalent directories are determined according to the rules set up by the user in the local and/or global map translation files. The format of these files is discussed in the next section.

The **-p** command line option specifies the values for both **rexecdir** and **rworkdir**, overriding all other methods of specifying them (**-p** is discussed later in this chapter).

About Map Translation

As we've stated, map translation is a way of defining equivalences between local directory trees and directory trees on remote nodes. If your network presents a consistent view of a common file system (via NFS or AFS, for example), then you will not need to worry about map translation. On the other hand, if your networked file systems are not completely consistent—if a file system is mounted as `/home` on one system and as `/net/home` on another system, for example—then map files can be a great help in automating TCP Linda execution.

Basically, map files provide a way of saying, "When I run a program from directory X on the local node, always use directory Y on remote node R." Map translation occurs for both the execution directories (locations of executable files) and working directories on each node.

Map translation means that **ntsnet** translates local directories to their properly defined equivalents on a remote node before specifying an executable on (or copy an executable to) that remote node (executable file distribution is discussed later in this chapter). If map translation is enabled—as it is by default—but no translation is explicitly specified for a given directory and/or node (whether because its rules haven't been specified or no map translation file exists), the rule is simple: look for the same directory on the remote node.

If enabled, map translation occurs whether the local directories are specified explicitly (as when the full executable file pathname is given on the command line) or determined implicitly (using the `TSNET_PATH` for example). Thus, map translation will occur for both of the following commands:

`$ ntsnet /tmp/test24`

`$ ntsnet test24`

In the first case, ntsnet will translate the directory you've specifie`d, /tmp,` for each node where the application `test24` will run; if no explicit translation has been defined, then **ntsnet** will perform a *null translation* and look for the program `test24` in `/tmp` on each remote node as well.

For the second command, **ntsnet** will first determine the location of the `test24` executable on the local node, using `TSNET_PATH` (or its default value), and then translate that location for each remote node involved in the job.

If the **rworkdir** resource is set to `parallel` (the default value), then the current working directory is also subject to map translation.

The Map Translation File

ntsnet uses the local and global map translation files, `~/.tsnet.map` and `common/ lib/tsnet.map` (relative to the Linda tree), respectively, to determine the correspondences between directories on different nodes. The first matching entry is used to perform each translation. The map translation mechanism is extremely powerful and can be used to define equivalences among systems, whether or not their file systems are linked with NFS.

Map translation is a two-step process. Rather than having to specify the exact translation for every pair of hosts within a network, map translation allows you to specify two rules for each host, to and from a generic value, known as a *generic directory*. A generic directory is a string—often a directory pathname—used essentially as a key into the various entries in the map translation file. The generic directory serves as the target for specific local and remote directories as **ntsnet** attempts to translate them for use on various nodes.

Map translation file entries use the following commands:

mapto   Map a specific local directory to a generic directory.

mapfrom   Map a generic directory to a specific directory (usually on a remote node).
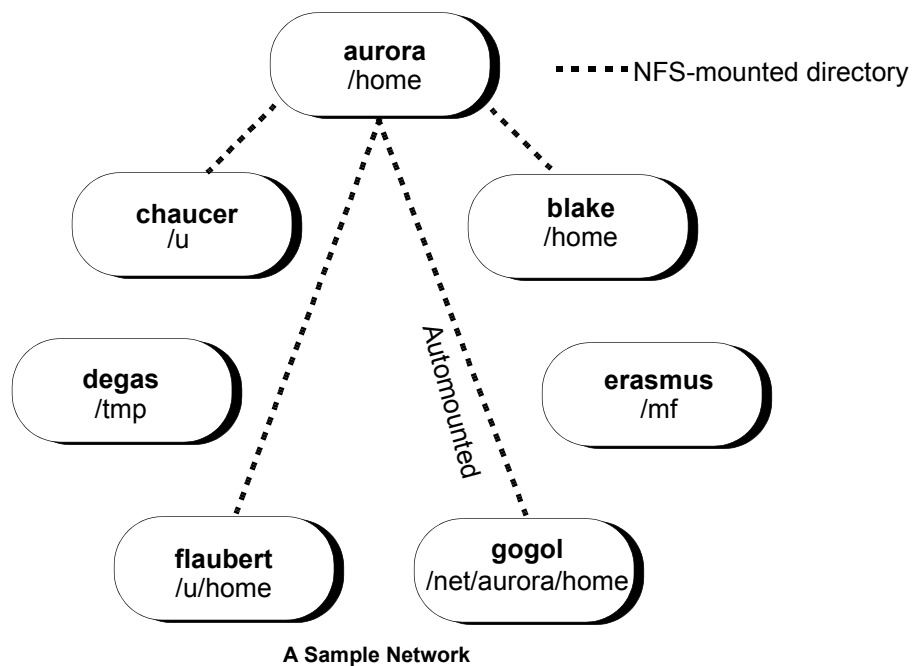
| **map** | Equivalent to a **mapto** and a **mapfrom**, mapping specific directories to and from a generic directory. |

Here is the general syntax for a map translation file entry:

```
mapverb generic-dir {
    node-name : specific-dir;
    node-name : specific-dir;
    ...
    }
```

The generic directory is always translated to a specific directory before being used as a location for executables or as the working directory on any node. Thus, there is no requirement that it even exist as a real directory path. In fact, it is possible to use any arbitrary symbolic name as a generic directory.

A Sample Network

These concepts will become clear once we look at several examples of map translation. We'll use the sample network shown in the preceding illustration, which shows the location of users' home directories for each node in the network, for our initial examples. On *aurora*, users' home directories are located in `/home`, the mount point for one of its local disks. This same disk is normally statically mounted via NFS to three other systems: *blake*, *chaucer*, and *flaubert*. On *blake*, it is also mounted at `/home`; on the other two systems, it is mounted at a different directory location (at `/u` on *chaucer*, and at `/u/home` on *flaubert*). Home directories on *erasmus* are found in the local directory `/mf`. On the node *gogol*, the home directories from *aurora* are automounted as necessary at `/net/aurora/home`. Finally, users don't generally have their own home directories on node *degas*; when they run remote worker processes on this node, they use `/tmp` as their working directory (which is what is listed in the diagram).

Consider the following command, run by user *chavez* from the `work` subdirectory of her home directory on *flaubert* (i.e., `/u/home/chavez/work`):

```
$ ntsnet test24g
```

For this command to work properly and successfully execute on all of the nodes in the sample network, we need to construct rules that tell **ntsnet** how to translate this working directory for each of the nodes. Most of the work can be done by this **map** entry, which uses `/home` as its generic directory:

```
map /home {

    chaucer : /u;

    erasmus : /mf;

    flaubert : /u/home;

    gogol : /net/aurora/home;

}
```

This entry translates the listed local directories to the generic directory `/home`. The translation applies to the entire tree starting at the specified locations. Thus, in our example, **ntsnet** will translate the (local) current working directory on *flaubert*, `/u/home/chavez/work`, to the generic directory `/home/chavez/work` (we don't have to explicitly construct a rule for the current directory). When it needs to determine the working directory for a remote process participating in the execution of the application `test24g`, it will use this generic directory. So, when it starts a remote worker process on *chaucer*, it will use the directory `/u/chavez/work` as the current directory, translating `/home` to `/u`, as specified in the rule for node *chaucer*. When **ntsnet** starts a remote worker process on *blake*, it still attempts to translate the generic directory, but no rule exists for *blake*. In this case, a *null translation* occurs, and the directory remains `/home/chavez/work, which is what is appropriate for this system.`

The rule we've written will allow us to run our **ntsnet** from the `work` subdirectory of *chavez*'s home directory on nodes *aurora* and *blake*, and on any of the listed nodes except *gogol*; in each case, the current working directory will translate to `/home/chavez/work`, which will in turn be translated to the appropriate directory when **ntsnet** starts remote worker processes.

If we want to run the **ntsnet** command on node *gogol*, however, we must create an additional rule. Home directories on *gogol* are automounted from *aurora* on demand. When referred to in the context of a process starting from a remote system, their location can be written as in the first rule; thus, when a remote process is initiated on *gogol* from *flaubert*, the current working directory for the remote node is correctly translated to `/net/aurora/home/chavez/work`.

However, if the **ntsnet** command is run instead on *gogol* from this same directory, the literal directory location—what is returned by **pwd** or the `getcwd` system call—is what is used for map translation, in this case, using the actual automounter mount point: `/tmp_mnt/net/aurora/home/chavez/work`. Thus, the translation from generic to remote directories is handled correctly by the first rule, and what is needed is a rule for translating this local directory to a generic directory. This is the function of **mapto** entries. The following entry maps the local directory on *gogol* to the same generic directory we've been using:

```
mapto /home {

    gogol : /tmp_mnt/net/aurora/home;

}
```

With this rule in place, running **ntsnet** from *gogol* will be successful, and the remote working directories we've considered so far will be set appropriately. Since this is a **mapto** entry, it will never be used to translate a generic directory to a working directory on *gogol*, an appropriate restriction given that automounted directories should not be explicitly referred to by their temporary mount points.

The last item we need to consider is translation rules for node *degas*. This node is a special case in that users do not have permanent home directories there, and are accordingly placed in the root directory when they log in. However, the system administrator feels that it is undesirable for remote processes to run from the root directory and wants them to run from `/tmp` instead. So, we need to equivalence the working directory we're using to `/tmp`. Unfortunately, given that there are no subdirectories under `/tmp` corresponding to the various users who might run remote processes on the system, we cannot write one simple rule to cover everyone. Instead, user *chavez* would need to include a rule like this one for her working directory within her local map translation configuration:

```
mapfrom /home/chavez/work {

    degas : /tmp;

}
```

Map Translation

The local directory `/u/home/chavez/work` *is translated to the generic directory* `/home/chavez/work`, *which in turn is translated to the specified remote directory for each listed remote node; a null translation occurs for nodes* `aurora` *and* `blake` *(as well as for any other node not mentioned in the map file), leaving the remote directory as* `/home/chavez/work`.
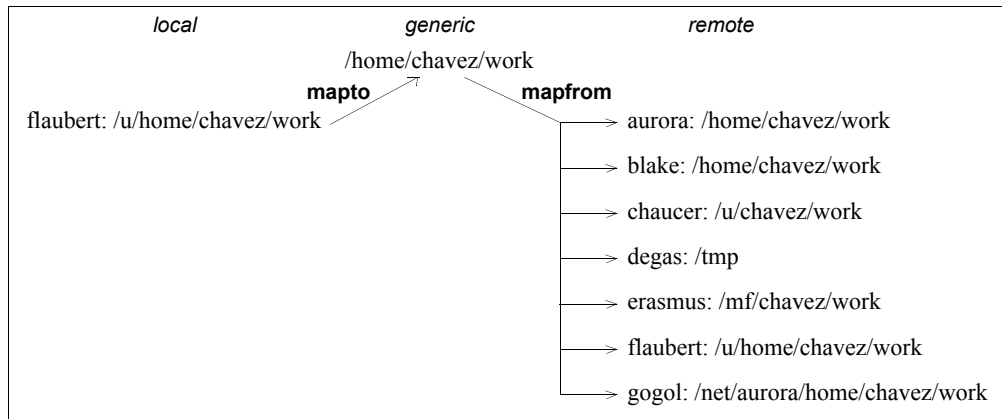
This particular rule facilitates generic-to-remote directory translation, enabling remote processes to be started on *degas* from any other node, using `/tmp` as the working directory. Note that the generic directory we've specified is the one to which her home directory will be translated from any other node. This rule will work only for user *chavez*; other users would need to construct their own versions.

This rule will not allow *chavez* to run **ntsnet** on *degas*, however, since it uses **mapfrom**. If it were necessary to be able to use *degas* as the local node, then the rule should be written as a **map**. The preceding diagram illustrates the map translation process using all three rules for our sample **ntsnet** command (executed on *flaubert*).

We've noted that generic directories need not exist as real directory locations. Here is a map file entry which takes advantage of this fact:

```
map special {

    chaucer : /u/guest/linda;

    blake : /home/guest1/appl/linda;

    sappho : /usr/guests/linda/bin;

    aurora : /public/bin;

    zola : /usr/local/bin;

}
```

`special` is not a real directory on any node; this entry defines the specified directories on the nodes listed as equivalent.

Map translation can be disabled by setting the **translate** resource (application-specific) to false (it is true by default), or by using the corresponding command line options (**-translate/+translate**). Remember also that map translation is used only for determining executable file locations on nodes for which the setting of **rexecdir** is `parallel`.

Similarly, the remote working directory is determined by map translation of the local (current) working directory only for nodes where the **rworkdir** resource is set to `parallel` (also its default value). In particular, map translation does nothing to filenames internal to the application.

Map Translation Entry Wildcards

There are two wildcard characters allowed in map translation file entries:

- An asterisk (`*`) can be used as a wildcard in local and remote node names.

- An ampersand (`&`) can be used to substitute the current node name within a translation.

For example, the following **mapto** entry handles the most common version of automounted directories:

```
mapto /net {

    * : /tmp_mnt/net;

}
```

It specifies the same remote directory for every remote node for the generic directory `/net`. The asterisk wildcard character can be used as above to represent the full node name. It can also be used as the initial component of a node name to specify wildcarded subdomains: `*.medusa.com`. No other placement of this wildcard character is supported. Here is an example entry using the ampersand wildcard:

```
mapto /net/& {

    sappho : /;

    blake : /;

}
```

These entries map the root directory to `/net/`*hostname* for the nodes *sappho* and *blake*. Thus, the local directory `/tests/new` on *sappho* would be mapped to the generic directory `/net/sappho/tests/new` by this entry.

The ampersand character can also be used within the actual translation specifications:

```
map working {

    * : /home/test/work/&;

}
```

This example equivalences the directories `/home/test/work/`*hostname* for all of the nodes within the network. Here is a more complicated example:

```
map /net/& {

    * : /;

    }
```

This entry maps the local root directory on all nodes to the generic directory `/net/`*hostname* (where *hostname* is replaced by the name of the local node). It also translates directories of that form back to their local equivalents when performing translation on remote directories, preventing unnecessary automounting.

When wildcarded entries produce multiple matches for a given translation, the longest matching string is used. When there are two matching strings of equal length, then the more specific match (i.e., containing fewer wildcards) is chosen.

Distributing Executables

ntsnet can distribute executable files prior to program execution. This is controlled by setting the **distribute** resource to true (application-specific). When different executables are required for the various remote systems (e.g., executables with different compilation options), all required executable files must be in the same directory as the local executable, and they will be copied to the target remote execution directories, as determined by the specific values set in the **rexecdir** resource for each node or by map translation. The **distribute** resource is false by default.

The **cleanup** resource (application-specific) determines whether remote executables are removed after the execution completes. Its default value is true. Note that the local executable is never removed, regardless of the setting of **cleanup**. The **cleanup** resource setting is relevant only when **distribute** is true.

The **distribute** resource may also be set with the **-distribute/+distribute** command line options (both abbreviable to **d**). The **cleanup** resource can be set with the **-cleanup/+cleanup** command line options.

Architecture–Specific Suffixes

By default, the names of executable files on all nodes are the same as the executable name given on the **ntsnet** command line. When executables are distributed prior to execution, this executable is the one that is copied by default.

However, it's possible that different flavors of the executable should be run on different nodes—**ntsnet** provides a mechanism for specifying which executable to use based on an architecture-specific suffix. For example, **ntsnet** can be told to use executables having the extension `.i686` on some nodes and to use ones with the extension `.i586` on others.[‡]

[‡] Using this mechanism, it's even possible to support accidental homogeneity—in which different machine architectures support the same data layout, but obviously, great care must be taken here.

These suffixes are used when the **suffix** resource (application-specific) is true (the default value). Which suffix to use for a given node is specified by the **suffixstring** resource (application and node specific). The default **suffixstring** value is the null string, so even though suffixes are used by default, this fact has no effect until some specific suffixes are defined using **suffixstring**.

Here is a section of a **ntsnet** configuration file illustrating a common use of these features:

```
Tsnet.Appl.Node.rexecdir: parallel

Tsnet.test24.nodelist: chaucer moliere sappho

Tsnet.test24.suffix: true

Tsnet.Appl.chaucer.suffixstring: .athalon

Tsnet.Appl.moliere.suffixstring: .pIII

Tsnet.Appl.sappho.suffixstring: .xeon

Tsnet.test24.sappho.suffixstring: .xOptBlasLib

Tsnet.Appl.aurora.suffixstring: .duron
```

These entries would result in the following executables being used for these nodes when running the application `test24` (located in whatever directory resulted from map translation):

chaucer	`test24.athalon`
moliere	`test24.pIII`
sappho	`test24.xOptBlasLib`
aurora (*local*)	`test24.duron`

If the **distribute** resource for the application `test24` is true, then files of these names will be copied to the remote nodes prior to execution. Otherwise, they must already exist there (in the proper directory).

The command line options **-suffix/+suffix** may also be used to specify the value of the **suffix** resource for the current run.

Specifying the Working Directory for Each Node

The working directory used for program execution on a remote node is also mapped from the current directory on the local node if the **rworkdir** resource (application and node specific) is set to `parallel` for that node. Alternatively, an explicit working directory can be set for an application, node, or application-node combination by giving an explicit directory as **rworkdir**'s value.

The **-p** option can be used to specify a remote directory for both **rexecdir** and **rworkdir** in a single option. It overrides any other method of setting either resource, including other command line options. Note that the old **tsnet** command requirement of including **-p** with `$cwd` as its option is no longer necessary. If the current directory is NFS mounted on all desired nodes, using the same path everywhere, then a command like this:

```
$ ntsnet hello_world ...
```

will work properly (even assuming no settings have been made via the **ntsnet** configuration file). It will run the executable `hello_world`, located in the current directory on all relevant nodes (by default, the nodes in the file `tsnet.nodes` in the current directory if no other provisions have been made), using the current directory as the default directory in each case. The **-p** option is no longer necessary in such cases.

Permissions and Security Issues

ntsnet assumes that all remote systems and directories are accessible. By default, it uses the local username for running remote processes. The **user** resource (node-specific) can be used to specify an alternate username on a remote system. For example, the following configuration file entry tells **ntsnet** to use the username *guest* when running process on node *sappho*:

```
Tsnet.sappho.user: guest
```

There is no provision for specifying passwords for use on remote systems. The standard TCP/IP mechanisms—the `/etc/hosts.equiv` file and individual user `.rhosts` files—should be used to assure access.

ntsnet Worker Process Scheduling

The TCP Linda System provides considerable control over how processes are scheduled on remote nodes. **ntsnet** uses the node list resources we've already looked at to determine the list of nodes where the application may potentially run. Whether a node is actually used depends on a number of other factors, which we'll examine in turn.

Forming The Execution Group

The list of potential nodes on which an application may run is determined by the **nodelist** resource; the set of nodes on which an application actually runs is called the *execution group*. **ntsnet** begins with the node set and successively starts remote processes using the scheduling rules described below until a sufficient number of them are running.

Technically, the processes started by **ntsnet** are known as *eval servers*. An eval server is a process started by **ntsnet** that waits to execute a Linda process. The process on the local node is known as the *master* in this context, and it too eventually becomes an **eval** server. The eval servers on remote nodes (and any additional eval servers on the local node) are known as *workers*. Note that this terminology is independent of the master/worker concepts within the application program. When used in this chapter, the terms master and worker refer exclusively to the initiating process on the local node and all other eval servers started by **ntsnet**, respectively.

How many worker processes are desired is controlled by the **maxworkers** and **minworkers** resources (application-specific). The default values are the number of distinct nodes in the **nodelist** resource (not counting the local node) for **maxworkers** and 1 for **minworkers**. The master attempts to start **maxworkers** workers when execution begins; if at least **minworkers** eventually join the execution group, the job proceeds. Otherwise execution terminates.

More specifically, execution will begin according to the following rules:

- ♦ **ntsnet** will attempt to start up to **maxworkers** worker processes. Until the time specified by the **minwait** resource, execution will begin immediately whenever **maxworkers** workers have joined the execution group .

- ♦ When the **minwait** interval has elapsed, if at least **minworkers** have joined the execution group, execution will start.

- ♦ Otherwise, **ntsnet** will continue trying to create workers until the **maxwait** interval has elapsed (which includes the time already spent in **minwait**). As soon as **minworkers** workers have started, execution will immediately commence.

- ♦ Once **maxwait** seconds have passed and the execution group is still smaller than **minworkers**, the startup process will fail, and execution will not proceed.

The default value for both **maxwait** and **minwait** is 30 seconds. The values for **maxwait** and **minwait** may also be specified using the **-wait** command line option. Its syntax is:

-wait *minwait*[**:***maxwait*]

For example, `-w 30:60` would set **minwait** to 30 seconds and **maxwait** to a total of 60 seconds. If **-wait** is given only a single value as its parameter, then that value is used for both resources.

Similarly, the values for **maxworkers** and **minworkers** may also be specified using the **-n** command line option. Its syntax is:

-n *minworkers*[**:***maxworkers*]

For example, `-n 2:4` would set **minworkers** to 2 and **maxworkers** to 4. If **-n** is given only a single value as its parameter, then that value is used for both **maxworkers** and **minworkers**.

Once **ntsnet** has attempted to start a process on a node, it waits for a message from the worker indicating that it has joined the execution group. Only after the master receives the join message is the worker added to the execution group and counted toward the minimum and maximum number of workers.

The following table summarizes the preceding information. At a given time t, having received p join requests, the master process will act as follows:

States & Outcomes	$p <$ **minworkers**	**minworkers** $\leq p <$ **maxworkers**	$p >$ **maxworkers**
$t <$ **minwait**	*wait*	*wait*	*success*
minwait $\leq t \leq$ **maxwait**	*wait*	*success*	*success*
$t >$ **maxwait**	*failure*	*success*	*success*

Selecting Nodes for Workers

Table 1 lists the resources used to determine whether a node is used for a worker process. When it wants to start a new worker process, **ntsnet** determines which node to start it on in the following manner. First, it calculates a new adjusted load for each node assuming that the process were scheduled to it, using the following formula:[†]

$$\text{(initial\_load} + (N_{master} * \text{masterload}) + (N_{worker} * \text{workerload})) / \text{speedfactor}$$

This quantity represents a load average value corrected for a variety of factors. The various components have the following meanings:

initial_load   If the **getload** resource is true (its default value), this is the load average obtained from the node (over the period specified in the **loadperiod** resource). If the remote procedure call to get the load average fails, the value in the **fallbackload** resource is substituted. If **getload** is false, then *initial_load* is set to `0`.

One potential use of the **fallbackload** resource is designed to prevent attempts to start new workers on nodes that have gone down. If the RPC to get the load average for a node fails and **fallbackload** is set to a large value, then it will be quite unlikely that the node will be chosen by the scheduler. Setting **fallbackload** to a value greater than **threshold * speedfactor** will ensure that it is never chosen.

N_{master} `1` if the master process is running on the node, `0` otherwise.

[†]   **ntsnet** makes use of **rstatd** to gather this information. If this service is not provided on a node, default values are assumed.

masterload	The second term in the numerator of the formula for the adjusted load means that the value of the **masterload** resource is added to the raw load average if the master process is running on the node. This enables an estimate of how the CPU resources the master will eventually use to be included in the adjusted load average (even though it is not currently consuming them). Set **masterload** to a smaller value than its default of 1 if it does not consume significant resources during execution (for example, if it does not itself become a worker).
$N_{workers}$	The number of worker processes already started on the node.

Table 1. Process Scheduling Resources

Resource	Meaning	Default Value	Equivalent ntsnet Option(s)	Scope
maxprocspernode	Maximum number of processes that can be run on any node. Includes the master process on the local node.	1	**-maxprocspernode** *n* **-mp** *n*	application-specific
getload	Whether or not to use current system load averages when scheduling workers on nodes.	true	**-getload/+getload**	application-specific
loadperiod	The period in minutes over which to compute load averages (when they are used for scheduling).	5	**-loadperiod** *mins* **-m** *mins*	application-specific
threshold	Maximum load allowed on a node; if the normalized load exceeds this value, then no worker will be started.	20		node-specific
speedfactor	A number indicating relative CPU capacity compared to other nodes. Larger values indicate increased ability to run multiple workers. Used in computing the adjusted load average.	1.0		node-specific
masterload	Load that the master process puts on its node (the local node). Used in computing the adjusted load average.	1	**-masterload** *n*	application-specific
workerload	Load that a worker process puts on its node. Used in computing the adjusted load average.	1	**-workerload** *n*	application-specific
fallbackload	Value to use if **ntsnet** is unable to obtain the current system load average. Setting this resource to a large value will ensure that nodes that are down will be excluded.	0.99	**-fallbackload** *n*	application-specific
available	Whether a node is available or not; useful for temporarily disabling a node without removing it from existing node sets.	true		node-specific

workerload　　The third term of the numerator adjusts the load average for the number of workers the node already has running, even though they are not yet consuming substantial CPU resources. You can alter this value from its default of 1 to reflect your estimate of how much load a single worker process adds.

speedfactor The **speedfactor** resource's value is used to normalize the corrected load averages for differing CPU speeds—and hence different total capacities—on different nodes. Generally, a value of 1 is given to the slowest system type within the network. Faster systems will then be assigned correspondingly higher **speedfactor** values. Multiprocessor systems are also given higher **speedfactor** values, generally set to the value appropriate for a single CPU multiplied by the number of processors.

Once all the adjusted loads are calculated, **ntsnet** finds the node having the lowest value, which presumably will be the most lightly loaded when execution begins. It then checks its adjusted load against the value of its **threshold** resource. If the adjusted load doesn't exceed it, **ntsnet** next checks the number of processes already started. If this number is less than the value of the **maxprocspernode** resource, another process is started; otherwise, **ntsnet** continues on to the next least heavily loaded node. This scheduling process continues until **maxworkers** processes are started or until no qualifying node can be found. The goal of the scheduling process is to minimize the load on the maximally loaded node.

Here are some sample **ntsnet** configuration file entries showing the uses of these resources:

```
Tsnet.Appl.getload: true

Tsnet.Appl.loadperiod: 10

! use the default speedfactor of 1 for these systems

Tsnet.Node.slowmach: priestley pinter

! fastguy is 5.25X slowmach

Tsnet.Appl.fastguys: sand stahl

! goethe has 2 heads and gogol has 4; each is 1.5X slowmach

Tsnet.Node.multiprocs: goethe gogol

Tsnet.Appl.nodelist: @slowmach @fastguys @multiprocs

! scheduler parameters

Tsnet.Appl.masterload: .5

Tsnet.Appl.workerload: 1

! maxprocspernode is set high so gogol can get a lot

Tsnet.Appl.maxprocspernode: 8
```

```
Tsnet.stahl.speedfactor: 5.25

Tsnet.sand.speedfactor: 5.25

Tsnet.goethe.speedfactor: 3

Tsnet.gogol.speedfactor: 6
```

This file attempts to set **speedfactor** values for the various nodes reflecting their relative CPU capacities. The **maxprocspernode** resource is set to a high value so that the multiprocessor system with 4 CPU's can run up to two workers per processor.

Special Purpose Resources

ntsnet provides several resources for use with various optional features which will be described in this section.

Tuple Redirection Optimization

By default, the TCP Linda system uses *tuple redirection*, an optimization designed to improve performance by detecting patterns in tuple usage and attempting to place tuples on those nodes where they will eventually be used. If successful, this optimization produces significant communications savings. This feature can be disabled by setting the value of the **redirect** resource (application-specific) to false (its default value is true). The value of this resource may also be set with the **-redirect/+redirect** command line options.

Disabling Global Configuration Files

The **useglobalconfig** and **useglobalmap** resources (both application-specific) specify whether to use the entries in the global configuration file and global map file in addition to the local files. In any case, local file entries take precedence over those in the global files. The default value for both resources in true. Command line options are available for both resources: **-useglobalconfig/+useglobalconfig** and **-useglobalmap/+useglobalmap**.

Generating Additional Status Messages

ntsnet can optionally display informational messages as program execution proceeds. Whether and how frequently messages are displayed are controlled by the **verbose** and **veryverbose** resources (both application-specific). Both are mainly useful for debugging configuration and map files, and both default to false. The command line options **-verbose/+verbose** (abbreviable to **-v/+v**) and **-veryverbose/+veryverbos**e (or **-vv/+vv**) can also be used to specify these resources.

Process Initiation Delays

ntsnet initiates worker processes by running the **rsh** command in the background on the local node, creating each successive process as soon as the previous command returns. Under some unusual network circumstances, such a procedure can overwhelm a network server process and result in errors. The **delay** resource is provided to handle such situations. It specifies the amount of time to pause between successive **rsh** command

initiations, in milliseconds (the default value is 0). If you experience such problems, try setting its value to 1000 (1 second). The **-delay** command line option may also be used to specify the value for this resource.

Appropriate Granularity for Network Applications

A network environment often has a significantly different granularity profile from other parallel computing environments (such as shared or distributed memory multiprocessors) because its communications speed is so much slower. This is the result of the differing communications bandwidth achievable via Ethernet and typical hardware interconnects in parallel computers. In general, networks impose relatively high communications overhead, and parallel programs need to be relatively coarse-grained to achieve good performance.

On a typical 100mbs Ethernet network, an **out/in** combination takes approximately 500 microseconds, and maximum throughput is about 10MB per second. Thus to **out** and then **in** a tuple of size S takes about (S/10000000) + .0005 seconds. So a work task that invovled 20000 bytes total of input and output data, should run at least 3milliseconds in order to justify the communication time, while a task involving a few bytes of input and output data should run at least 1 millisecond.

As new network interconnect technology is developed, the granularity effects must be re-examined. The use of certain high speed switches, for example, may give networks performance characteristics almost identical to distributed memory parallel computers.

Forcing an eval to a Specific Node or System Type

While there is no way within an **eval** operation itself to force its execution on any particular node or type of system, adding an additional function layer in front of the target routine can accomplish this. Here is an example:

```
master()
for (i=0; i < NWORKERS; i++)
     eval("worker", do_worker());

do_worker()
{
   get the hostname or architecture type via standard system call
   if (! strcmp(host, "moliere")) worker_1();
   elseif (! strcmp(host, "goethe")) worker_2();
   elseif (! strcmp(arch, "p4")) p4_worker();
   and so on
}
```

The **eval** operation calls a generic worker function, `do_worker`, which determines the hostname (or architecture type), and then calls the appropriate real worker function.

Debugging TCP Linda Programs

There are two ways of debugging a TCP Linda program:

- ◆ Use the **ntsnet debug** option.
- ◆ Manually start program execution on each node.

This section looks at each method in turn; both discussions assume that you have properly prepared executables for use with a debugger by compiling them with **-g**.

ntsnet's Debug Mode

The **-debug** option to the **ntsnet** command initiates TCP Linda program execution in debug mode. Including this option on the command line starts an **xterm** process running the debugger specified by the **debugger** resource on each participating node. The value for the **debugger** resource defaults to **dbx**; the other currently-supported debugger is **gdb** (the debugger from the Free Software Foundation). Setting the value for **debugger** to **none** effectively disables debugging on a particular node, as in this example:

```
Tsnet.moliere.debugger: none
Tsnet.Node.debugger: gdb
```

The second configuration file command sets the default value for the **debugger** resource to **gdb**, while the first line prevents debugging on node *moliere*. The **debugger** resource is node-specific.

For example, the following command will create three debugging windows executing the program **test24** on nodes selected from the **ntsnet** node list in the usual way:

```
$ ntsnet -debug -n 2 test24
```

The node and application name will appear in the title bar of each window.

Once all of the debugger processes have started up, you can set breakpoints, run in single step mode, examine variables, and perform all other normal debugging functions for each process. **ntsnet** facilitates program initiation by defining the alias **lrun** within each debugging session to be the appropriate command line to begin application execution. You should use this alias, rather than the debugger's normal program initiation command (e.g., **run** in **dbx).**

Once the program has finished executing, the controlling (master) process will exit and the debugger prompt will appear in the corresponding window. However, the other (worker) processes will not return. To terminate all program processes, enter the **quit** command to the debugger for the master process, and **ntsnet** will automatically terminate all of the other processes.

The **debug** resource (application-specific) can be used instead of the command line option. A value of true is equivalent to including the command line option (the default value is false).

Hints on running TCP Linda programs in Debug Mode

Keep in mind that each process is handling a portion of tuple space in addition to running the application program. Therefore, when a process is paused—for example, at a breakpoint—then no tuple space requests can be handled by it. For this reason, it's best to break only a single process at a time, with all other processes either continuing or stepping through a (blocked) **in** or **rd** operation.

- ◆ **ntsnet** relies on the command search path being set appropriately on all remote nodes. Specifically, the locations of **xterm**, the debugger, **sh**, and **rcp**(**scp**) need to be in the search path. Note that remote TCP Linda processes are initiated with **rsh** (not **rlogin**). Hence, make sure that the PATH environment variable is set properly even if the login initialization file is not executed. You can test this by running **rsh** manually, and you can ensure this by placing the variable's definition in ~/.cshrc rather than ~/.login if you use the C shell.

- ◆ In some network configurations, it can be necessary to give remote hosts access to the local X server. This is the purpose of the **xhost** command. You will need to run **xhost** if you see error messages like this one:

  Xlib: Client is not authorized to connect to Server

  If it is required, you can execute the **xhost** command manually. For example, the form **xhost +** grants access to the local X server to all remote hosts. You can also specify a list of nodes (check the man page for details).

  Alternatively, you can modify the linda_rsh shell script (located in the bin subdirectory of the Linda tree), adding the option **-access** to the **xon** command, which causes the latter script to run **xhost** automatically.

- ◆ **ntsnet**'s debug mode also changes the default values of some other resources:

  - • The **workerwait** resource defaults to 1000000 seconds.
  - • The **maxwait** resource defaults to 1000000 seconds.
  - • The **nice** resource is overridden to be false.

- ◆ **ntsnet** ensures a consistent **dbx** environment across all nodes by copying the .dbxinit file from the local node to all participating remote nodes. The **dbx** command **ignore IO** is also often useful when debugging TCP Linda programs. This is not necessary for **gdb**.

Running TCP Linda Programs Without ntsnet

TCP Linda programs can also be executed manually by invoking **ntsnet** with the
-launchByHand option. These are the steps for doing so:

- ◆ Establish a session on each desired remote node (via **rlogin** for example). It will
  be most convenient to start each discrete session in a separate window. In general,
  start as many processes as you would when executing the program with **ntsnet**. If
  you plan to use a debugger, start it on the desired remote nodes.[‡]

- ◆ Run **ntsnet -launchByHand ...** in a separate session.

- ◆ In the session where you will start the master process, make sure that the
  LINDA_PATH environment variable is set correctly. It should point to the
  top-level Linda installation directory, and the directory specification must include
  a final slash.

- ◆ Begin program execution using the commands output by **ntsnet** (cut and paste
  will reduce errors), one per remote session. These have the general form:

**Master
session**
    application [*appl-arguments*] +LARGS [*linda-args*]

**Worker
sessions**
    node-name application +LARGS [*linda-args*]

where *application* is the program command, *appl-arguments* are the application's
arguments (if needed), *linda-args* are any TCP Linda run-time kernel options, and
node-name is the name of a remote node.

- ◆ Keep in mind that each process is handling a portion of tuple space in addition to
  running the application program. Therefore, when a process is paused—for
  example, at a breakpoint—then no tuple space requests can be handled by it. For
  this reason, it's best to break only a single process at a time, with all other
  processes either continuing or stepping over an **in** operation.

[‡] To do so effectively, you need to have compiled the application with **-g**.

4
Case Studies

This chapter provides detailed discussions of several real applications parallelized with Linda. In some cases, the descriptions will follow the programming process recommended in Chapter 1 by first transforming a sequential program into a parallel program in a straightforward manner and then, if necessary, optimizing it to produce a more efficient parallel program; in others, we'll focus on topics and issues of special importance. Space limitations will not allow for a full treatment of any of these programs, but these case studies illustrate many important techniques useful to any programmer working with Linda.

Ray Tracing

Image rendering is a very computationally intensive application. The **Rayshade** program, written in C, renders color images using ray tracing. It is capable of including texturing effects to simulate different materials in the image, multiple light sources and types (point, spotlight, diffuse), and atmospheric effects like fog or mist. **Rayshade** computes the effects of all of these factors on the image, also taking into account reflections and shadows.

This case study illustrates the following techniques:

- ◆ Dividing the main task among the workers.
- ◆ Adapting the sequential program structure to the master/worker paradigm.

Rayshade's `main` routine is shown below. (As mentioned in the Introduction, we've removed some sections of code and replaced them with descriptive comments, and we've ignored others, like declarations, altogether.)

```
main(argc, argv)
{
    Setup.
    parse_options(argc, argv);
    read_input_file();
    Initialization.
    startpic(); /* start new picture */
    More setup.
    raytrace();
}
```

After some initial setup, **Rayshade** processes the command line options, validating them for correctness and determining which options have been selected. It then reads in the image and then performs some additional initialization steps.

Next, **Rayshade** calls the function `startpic`, which logically creates a new picture, followed by some additional setup activities. Finally it calls `raytrace` which initiates the actual rendering.

The bulk of the work is handled by the routine `do_trace`, called by `raytrace`:

```
do_trace()/* called by raytrace */
{
   Set sampling parameters.
   /* Trace each scanline, writing results to output file. */
   for (y = StartLine; y >= 0; y--) {
      trace_jit_line(y, out_buf);
      outline(out_buf);
   }
}
```

After testing and setting parameters controlling the sampling for this rendering operation, `do_trace`'s main loop executes. For each scan line in the final image, it calls `trace_jit_line` and `outline`. The former handles the ray tracing computations, through many subordinate routines, and `outline` writes the completed line to a file.

This sequential program already divides its work into natural, independent units: each scan line of the final image. The parallel version will compute many separate scan lines in parallel. We'll look at how **Rayshade** was restructured to run in parallel in some detail.

To begin with, a new `real_main` routine was created. This is not always necessary; sometimes it is best just to rename the existing main to `real_main`. In this case, `main` was renamed `rayshade_main`, and `real_main` calls it. This was done because the original `main` routine needs to be called by each worker process, as we'll see.

Here is `real_main`:

```
real_main(argc, argv)
{
   for (i = 0; i < argc; ++i)
      strcpy(args[i], argv[i]);
   out("comm args", args:argc);
   return rayshade_main(argc, argv, 0);
}
```

`real_main`'s tasks are very simple: place a copy of the command line arguments into tuple space—accomplished by copying them to a local array, which is then used by the **out** operation—and then call `rayshade_main` with its original arguments and one additional new one.

Here is `rayshade_main` (the additions made for the C-Linda version are in boldface):

```
rayshade_main(argc, argv, worker)
{
   Setup:
   parse_options(argc, argv);
```

```
   read_input_file();
   Initialization:
   if (worker)
      return;

   startpic();     /* Start new picture */
   More setup:
   raytrace();
}
```

The third argument is a flag indicating whether the caller is a worker process or not (a value of 1 means it is a worker). This flag is used to ensure that the worker exits at the proper time, and the remaining initialization steps are executed only at the beginning of the job.[†]

A few lines were also added to `parse_options` to handle additional options specifying the number of workers to use for the parallel version.

`rayshade_main` still ends by calling `raytrace`. No changes were made in that routine, but a lot was done to the support routine `do_trace` that `raytrace` calls.

`do_trace` becomes the master process in the parallel version of **Rayshade**. It's important to note that the master does not always need to be the top-level routine `real_main` or even the original `main` function; any routine can become the master process.

Here is `do_trace`:

```
do_trace()
{
   out("JitSamples", JitSamples);
   out("TopRay", TopRay);
   for (i = 0; i< Workers; i++)
      eval("worker", worker());
   out("scaninfo", StartLine);
   for (y = StartLine; y >= 0 ; y--) {
      in("result", y, ? out_buf:);
      outline(out_buf);
   }
}
```

The new version begins by placing two parameters needed by lower level routines into tuple space. The worker processes will read them and pass them along to the routines that need them. The function then starts `Workers` worker processes, each running the routine `worker`.

[†] While this program structure (in which the workers each call the main routine) is unusual, the technique of having workers repeat the setup code is not, because it is often faster.

Once the workers have all started, `do_trace` creates the task tuple "scaninfo." As in the Freewake example, **Rayshade** uses a counter tuple to indicate which task—in this case, which scan line—should be done next. Here the counter is initially set to the maximum scan line number to be computed; each worker will decrement the counter as it grabs a task, and when the counter falls below 0, all tasks will be complete.

The second `for` loop gathers completed scan lines from tuple space, again counting down from the maximum scan line number, and sends each one to the output file. The lines need to be gathered in order so that they are placed into the file in the proper location; they may not arrive in tuple space in that order, however, so `do_trace` may spend some time blocked.

The last piece needed for the parallel version of **Rayshade** is the `worker` function:

```
worker()
{
    rd("comm args", ? args: argc);
    for ( i = 0; i < argc;++i)
        argv[i] = (char *)(args+i);
    rayshade_main(argc, argv, 1);
    rd("TopRay", ? TopRay);
    rd("JitSamples", ? JitSamples);
    Set sampling parameters.
    while (1) {
        in("scaninfo", ? y);
        out("scaninfo", y-1);
        if (y < 0)
            break;
        trace_jit_line(y, out_buf);
        out("result", y, out_buf:Xres);
    }
    return;
}
```

The worker begins by reading the command line arguments from tuple space into a local array with the **rd** operation. It then calls `rayshade_main` to perform the necessary initialization and setup activities. It is often just as fast to have workers each repeat the sequential program's initialization process rather than doing it only once and then attempting to transmit the results of that process to the workers through tuple space.

However, when you choose to have the worker repeat the program initialization, it is important not to repeat steps that must occur only once. Here, the workers call `rayshade_main` with a third argument of `1`, ensuring that picture initialization is not repeated.

The worker function next retrieves the global parameters from tuple space, and sets the sampling parameters, using the same code as originally appeared in `do_trace`.

Each iteration of the `while` loop computes one scan line of the final image and places it into tuple space for later collection by the master process (executing `do_trace`). It removes the "scaninfo" counter tuple from tuple space, decrements it, and puts it back so that other processes can use it. If the counter has fallen below zero, all scan lines are finished, the loop ends, and the worker returns (terminating its process).

If the counter is non-negative, the worker generates that scan line by calling the same routine used by the sequential program. However, instead of an immediate call to `outline`, the computation is followed by an **out** operation, placing the results into tuple space. Sometime later, `do_trace` will retrieve the line and then make the call to `outline` to write it to the output file. Meanwhile, the worker has started its next task. It will continue to perform tasks until all of them are done.

`worker` bears a striking resemblance to the original version of `do_trace`; it adds a call to `rayshade_main` unnecessary in the sequential version, and it lacks only the call to `outline` from its main loop (which remains behind in the master process version of `do_trace`). This separation results from the need to pass data between master and workers via tuple space. (It is also much easier to code than sending explicit messages between all of the various processes.)

Matrix Multiplication

Matrix multiplication is a common operation central to many computationally intensive engineering and scientific applications. We've already looked at matrix multiplication in a general way in Chapter 1. Here we will look at a couple of approaches to parallel matrix multiplication, and conclude by considering when one should—and should not—use them. This example again uses C-Linda.

This case study illustrates the following techniques:

- ◆ Adjustable granularity and granularity knobs.
- ◆ Deciding where in a program to parallelize.

The basic matrix multiplication procedure is well-known: multiplying an L by M matrix A by an M by N matrix B yields an L by N matrix C where C_{ij} is the dot product of the ith row of A and the jth column of B. We won't bother translating this procedure into a sequential C program.

Instead, let's look at a simple, straightforward parallel matrix multiplication. Here is the master routine:

```
real_main()
{
   read_mats(rows, columns, L, M, N);

   for (i=0; i< NWORKERS; i++)
      eval("worker", worker(L, M, N));

   for (i=0; i < L; i++)
```

```
        out("row", i, rows[i]:M);
    out("columns", columns:M*N);

    out("task", 0);

    for (i=0; i < L; i++)
        in("result", i, ?result[i]:len);
}
```

The function begins by reading in the matrices (we're ignoring the details of how this happens). Next, it starts the worker processes, places each row of A and the entire B matrix into tuple space, and then creates the task tuple. Finally it collects the rows of C as they are completed by the workers.

The worker process begins by **rd**ing the B matrix from tuple space. After doing so, it enters an infinite loop from which it will exit only when all tasks are completed.

Each task consists of computing an entire row of C, as specified by the task tuple. The worker process retrieves this tuple, increments its value, and places it back into tuple space, checking to make sure it is not already greater than the maximum number of rows to be computed:

```
worker(L, M, N)
{
    rd("columns", ?columns:len);

    while (1) {
        in("task", ?i);
        if (i >= L) {
            out("task", i);
            break;
        }
        else
            out("task", i+1);

        rd("row", i, ?row:);

        for(j=0; j < N; j++)
        result[j]=dot_product(row, columns[j*M], M);

        out("result", i, result:N);
    }
    return;
}
```

The worker next reads in the row of A that it needs. Its final loop computes each element of the corresponding row of C by forming the dot product. When all of its entries have been computed, the worker sends the completed row of C to tuple space and begins the next task.

Unfortunately, this version has a disadvantage that not only inhibits it from always performing well, but sometimes even prevents it from running at all. It assumes that each worker has sufficient local memory to store the entire B matrix. For large matrices, this can be quite problematic. For such cases, we could modify the master to place each column of B into a separate tuple, and have the workers read them in for each row of A, a column at a time, but this would add significant communications overhead and probably yield poor performance. Instead, a more flexible program is needed where the amount of work done in each task can be varied to fit memory limitations and other restrictions imposed by the parallel computing environment.

Here is a version which fulfills this goal:

```
master(L, M, N)
{
    Allocate memory for matrices.
    Read in matrices.

    /* put matrices into tuple space */
    for (i = 0; i < L; i += clump,
            rows += M*clump, columns += M*clump) {
        out(i, "rows", rows:M*clump);
        out(i, "columns", columns:M*clump);
    }

    /* start workers and make first task */
    for (i=0; i < workers; ++i)
        eval("worker", worker(L, M, N, clump));

    out("task", 0);

    for (i = 0; i < L; i += clump, result += M*clump)
    in("result matrix", i, ? result:);
}
```

The master begins by allocating memory for and reading in the matrices. Next, it places the matrices into tuple space in chunks, each of which is clump rows or columns[‡] of its matrix. clump functions as a granularity knob in this program: a parameter whose value determines the task size, at least in part. Changing clump's value directly affects how large each chunk is.

The master then starts the workers and creates the task tuple as before. Its final loop retrieves the result matrix (C) from tuple space, again in chunks of clump rows. Code to reassemble C from these chunks is omitted.

[‡] Both versions of this program assume that A is stored by row in the rows array, and B is stored by column in the columns array. Such a strategy makes accessing the proper elements of each one much easier.

Here is the corresponding worker function:

```
worker(L, M, N, clump)
{
    Allocate memory for matrix chunks.
    while (1) {
        in("task", ? rows_index);
        if (rows_index < L)
            out("task", rows_index + clump);
        else {
            out("task", L);
            return;
        }

        rd(row_index, "row", ?row:);

        for (col_index = 0; col_index < N; col_index += clump) {
            rd(col_index, "columns", ? columns:);
            for (r = 0; r < clump; ++r) {
                result_ptr = result + ((r + col_index) * M);
                for (c = 0; c < clump; ++c){
                    dot = 0;
                    rp = rows + (r * M);
                    cp = columns + (c * M);
                    for (i = 0; i < M; ++i, ++rp, ++cp)
                        dot += *rp * *cp;
                    *result_ptr++ = dot;
                }
            }
        }
        /* Put a block of the result matrix into tuple space */
        out("result", rows_index, result:M*clump);
    } /* end while */
} /* end worker */
```

The worker begins by retrieving a task (and exiting if all tasks are already done). It reads the corresponding tuple from tuple space (containing `clump` rows of A). It then reads the columns of B, in chunks of `clump` columns at a time (as they have been partitioned into tuples). For each chunk, all corresponding elements of the result matrix are computed; when all of the chunks of the columns array (holding B) have been read and processed, the worker sends the completed chunk of the result matrix to tuple space and begins the next task.

The variable `clump` can get its value in any number of ways: from a preprocessor variable, from a command line argument, as some function of the sizes of the matrices, and so on. It allows you to adjust program execution in a number of ways. For example, if local memory for worker processes is limited, `clump` could be chosen so that the portions of A and B a worker held at any given time (`2*clump*M` total elements) would fit within the available amount. Or if the program were running on a network containing processors of greatly differing speeds, then it might sometimes be preferable to make the

task size smaller so that there are enough total tasks to keep every processor busy for essentially the entire execution time (with faster processors completing more tasks in that time). Building such adjustability into a program is one way to ensure easy adaptability to different parallel computing environments.

Even this version makes some optimistic assumptions, however. For example, it assumes that the master process has sufficient memory to read in both matrices before sending them to tuple space. If this assumption is false, then the disk reads and **out** operations would need to be interleaved to minimize the amount of data the master has to hold in local memory.

We'll close this consideration of parallel matrix multiplication by looking at a case where one might not want to use it. Here is a portion of a subroutine from a computational chemistry program (written in Fortran):

```
Subroutine gaus3(x,n)

Loops containing many independent exponentials
call matrix_multiply(chi,coo,psi)
call matrix_multiply(chix,coo,gx)
call matrix_multiply(chiy,coo,gy)
call matrix_multiply(chiz,coo,gz)
call matrix_multiply(chid2,coo,d2)
Return
End
```

Here are five independent matrix multiplication operations. Certainly, it would be nice to produce a parallel version of this program, but the question is: where to parallelize? We could replace each matrix multiplication call with a parallel version, or execute all five calls simultaneously. In some circumstances, each of these solutions will be the right one.

Here, however, we'd also like to execute some of those `exp` calls in parallel as well, which would tend to favor the second approach: creating a parallel version of `gaus3` that might very well use a sequential matrix multiplication routine.

There is a third possibility as well. `gaus3` is itself called a large number of times:

```
    Do 10 I=1,npts
        call gaus3(x,m)
 10    Continue
```

and again the calls are independent of one another. It might be possible to execute some of these calls in parallel, leaving `gaus3` essentially untouched. Whether this is a good idea or not depends on the likely value of the loop's upper limit, `npts`. If `npts` is typically, say, 8, then parallelizing at this point will limit the number of processor which the program could take advantage of to 8, and so it is probably better to parallelize `gaus3` itself. If, on the other hand, `npts` is typically 500000, then this is a perfectly good place to focus attention, and the job will be much simpler as well.

Database Searching

This case study looks at a generalized database search program. What the particular data records are is less important than the general issues all such searches raise, many of which are applicable to other types of problems as well. Readers wanting a more rigorous and specific treatment of this topic should consult Chapters 6 and 7 of *How to Write Parallel Programs* by Carriero and Gelernter.

This case study discusses the following techniques and issues:

- Distinct task tuples.
- Using watermarking to avoid overwhelming tuple space.
- Task ordering to aid in load balancing.
- Dealing with unequal task sizes.

Here is a simple sequential program (in Fortran) to search a database:

```
      Program DB_Search

      Call Get_target(target)
10    Call Get_next(DB, rec)
      If (rec .EQ. EOF) go to 100
      Call Compare(target,rec,result)
      Call Process(result)
      Goto 10

100   Continue
```

This program compares a target record (or key) against records in a database. The following discussion assumes that the operation of comparing two data records takes a substantial amount of time. Many such database applications exist, including ones designed for DNA sequence searching, credit application retrieval, and many other sorts of complex string matching.

The program hides all the details of the operation in its component subroutines: `get_next_record` retrieves another record from the database, `compare` compares the record to the target, and `process` takes `compare`'s result and keeps track of which records have matched (or come closest to matching) so far. When all relevant records have been tested, `output` will print the final results.

This version could search any kind of database, given appropriate versions of `get_next_record`, `compare`, and `process`. `get_next_record` could be implemented to return every record in the database in sequential order, or according to some sorting criteria, or it could return only selected records—those most likely to match the target, for example (think of searching a database containing fingerprints).

compare might return a yes or no answer, depending on whether target matched the current record or not, or it might return some value indicating how close a match the two were. In the first case, process would only need to keep track of positive results—matches—while in the second it would probably want to report on some number of best matches at the conclusion of the program.

Transforming this program into a parallel version is fairly straightforward. Each task will be one comparison. This time, we'll use one tuple for each task, holding the actual database record, rather than a single counter tuple. Here is the parallel master routine:

```
        Subroutine real_main

        Do 10 I=1,NWORKERS
            eval('worker', worker())
10      Continue
        Call Get_Target(target)
        out('target', target)

        NTasks=0
20      Call Get_Next(DB,Rec)
        IF (Rec .EQ. EOF) Go TO 30
        out('task', rec, OK)
        NTasks=NTasks+1
        Goto 20

30      Do 40 I=1,NTasks
            in('result', ?res)
            Call Process(res)
40      Continue

        DO 50 I=1,NWORKERS
            out('task', dummy, DIE)
50      Continue
        Return
        End
```

This program first starts the workers, gets the target, and places it into tuple space. Then it loops, retrieving one record from the database and creating a corresponding task tuple until there are no more records. Then it retrieves the results generated by the workers from tuple space and hands them to process.

Finally, in its final loop, the master process generates one additional task tuple for each worker. These tasks serve as *poison pills*: special tasks telling the workers to die. The task tuple's third field holds either the value represented by OK, meaning "this is a real task," or the one represented by DIE, meaning terminate.

Here is the corresponding worker:

```
        Subroutine worker

        rd('target', target)

        DO While(.TRUE.)
            in('task', rec, flag)
            If (flag .EQ. DIE) Goto 100
            Compare(rec, target, result)
            out('result', result)
        EndDo
100     Continue
```

The worker loops continuously, reading tasks and comparing records, placing the results into tuple space for the master to gather later, until it encounters the poison pill, at which point it exits.

Straightforward as this version is, it has some potential pitfalls. The most serious occurs if the database has large records, or large numbers of records, or both. In either case, the master could easily generate tasks much faster than the workers could complete them, and fill up tuple space in the process, causing the program to run out of memory and terminate.

A technique known as *watermarking* can provide protection against this eventuality. Watermarking involves making sure that there are no more than a fixed number of task tuples at any given time (the high water mark, so to speak). Once this limit is reached, the master process must do something else—such as gathering results—until the number reaches a lower bound (the low water mark), at which time it can go back to creating tasks. When the number of tasks once again reaches the upper bound, the process repeats.

Here is a version of the master process with watermarking:

```
20      Call Get_Next(DB,Rec)
        IF (Rec .EQ. EOF) Go TO 30
        out('task', rec, OK)
        NTasks=NTasks+1
        IF (NTasks .LE. UPPER_BOUND) Goto 20
        DO While(NTasks .GT. LOWER_BOUND)
            in('result', res)
            Call Process(res)
            NTasks=NTasks-1
        EndDo
        Goto 20

30      Do 40 I=1,NTasks
            in('result', ?res)
            Call Process(res)
40      Continue
```

Creating a new task increments the `ntasks` counter. Once it reaches its maximum value, the master switches over to gathering and processing results, decrementing the `ntasks` counter, which now holds the number of outstanding tasks, since every time the master finds a result tuple, it can be sure a task has been consumed. When the number of outstanding tasks reaches its minimum allowed value, the `do` loop ends, the master begins to make new tasks, and the process begins again.

After all needed tasks have been created, the master still needs to gather any remaining results from tuple space, which is the purpose of the second `while` loop.

`UPPER_BOUND` and `LOWER_BOUND` allow this program to adapt to its environment. Their values can be adjusted based on the size of tuple space, on the sizes of individual database records of the database as a whole, on the relative speeds of the `get_next_record`, `compare`, and `process` functions, and so on.

It can be as important to make sure that there are enough tasks in tuple space as to make sure there aren't too many. When there aren't enough tasks to keep all the workers busy, then *work starvation* sets in, and performance diminishes. Thus, if `LOWER_BOUND` were too low, there might be periods where workers would actually have to wait for their next task, a condition which is virtually never desirable.

Load balancing is another consideration that often comes into play. This program will perform fine if all of the comparisons take about the same amount of time, as would be the case when comparing fingerprints. However, there are many cases where different comparison operations take vastly different amounts of time—comparing DNA sequences, for example. In such cases, the program must ensure that the more time consuming comparisons do not become the rate limiting steps in the entire job. For example if the comparison which took the longest was started last, the other workers would all finish and sit idle waiting for it.

Sometimes, such problems can be avoided by paying attention to the order in which records are obtained, for example, by making `get_next_rec` more sensitive to task size in our example. This can complicate `get_next_rec` a great deal, to the point where it too would benefit from becoming a parallel operation. Of course, the same kinds of considerations hold for the routine `process` as well.

At other times, it is the comparison itself that needs to be parallelized. It may not be sufficient to perform several comparisons at once when an individual comparison takes a very long time.

In either of these cases, it will not be possible to take the generic approach to database searching that we have here. Rather, the specifics of the comparison or record retrieval or results processing algorithms will have to be examined explicitly, and creating a parallel version of one or more of them will be necessary to achieve good performance. In Chapter 7 of their book, Carriero and Gelernter present an elegant combination database searching program that parcels out small comparisons as a whole and divides large ones into discrete pieces.

Molecular Dynamics

Molecular dynamics calculations are performed to simulate the motion within molecules over time. This method is used to study the very large molecules of commercial and biological interest, typically containing thousands of atoms. The atoms in these molecules are constantly in motion; this movement results in changes in the overall molecular structure, which can in turn affect the molecule's properties. A molecular dynamics simulation calculates the changing structure of a molecule over time in an effort to understand and predict its properties. These calculations are carried out iteratively, solving for the total molecular energy and the forces on and positions of each atom in the molecule for each time step. In general, an atom's position depends on the positions of every other atom in the molecule, making molecular dynamics calculations require significant computational resources.
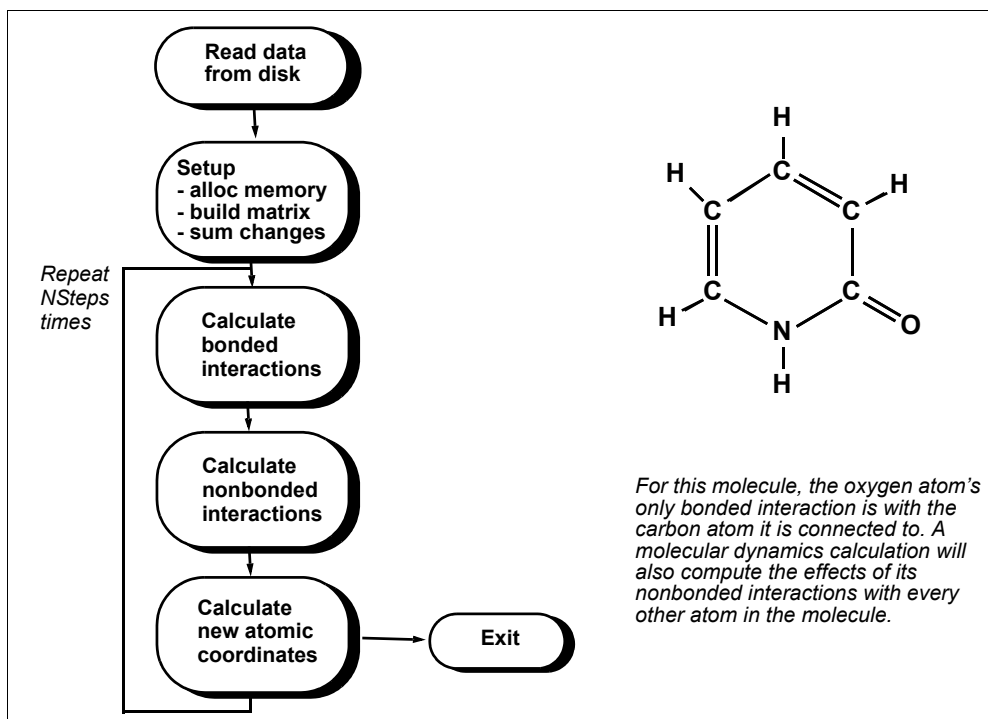
This case study illustrates the following techniques:

- ◆ Per-iteration worker wakeup.
- ◆ Cleaning up tuple space.
- ◆ Distributed master functions.

Since the original program for this case study is very long, we'll only look at the central portions as we examine how it was parallelized with C-Linda. This program required changes to several key routines, and illustrates using C-Linda to parallelize the calculation setup work as well as the computation core.

The diagram below presents a schematic representation of the sequential version of the computation:

After performing some initial setup steps in which it reads in and stores the data for the calculation and calculates the sums of special charges and some other quantities for the molecule, the program enters its main loop. For each time step—one loop iteration—the program must calculate the bonded and nonbonded interactions among all of the atoms in the molecule. The bonded interactions occur between atoms that are directly connected together by a chemical bond, and the nonbonded interactions are the effects of atoms that are not bonded upon one another's position. The latter take up the bulk of the time in any molecular dynamics calculations because they are both more numerous and more complex than the bonded interactions.

Sequential Version of the Molec. Dyn. Program & Bonded vs. Nonbonded Interactions

Here is a simplified version of the original `main` routine:

```
main(argc,argv)
{
   T = 300.0.
   process_args(argc,argv);
   Read data.
   Initialize parameters & data structures.

   verl_init(str,coor,vel,s);
   Te = temperature(str,kinetic_energy(str,vel));
   verl_scale(coor,sqrt(T/Te));

   for (i=0; i < NSteps; i++) {
      verl_step(str,coor,vel,s);
      Te = temperature(str,kinetic_energy(str,vel));
      verl_scale(coor,sqrt(T/Te));
   }
   exit(0);
}
```
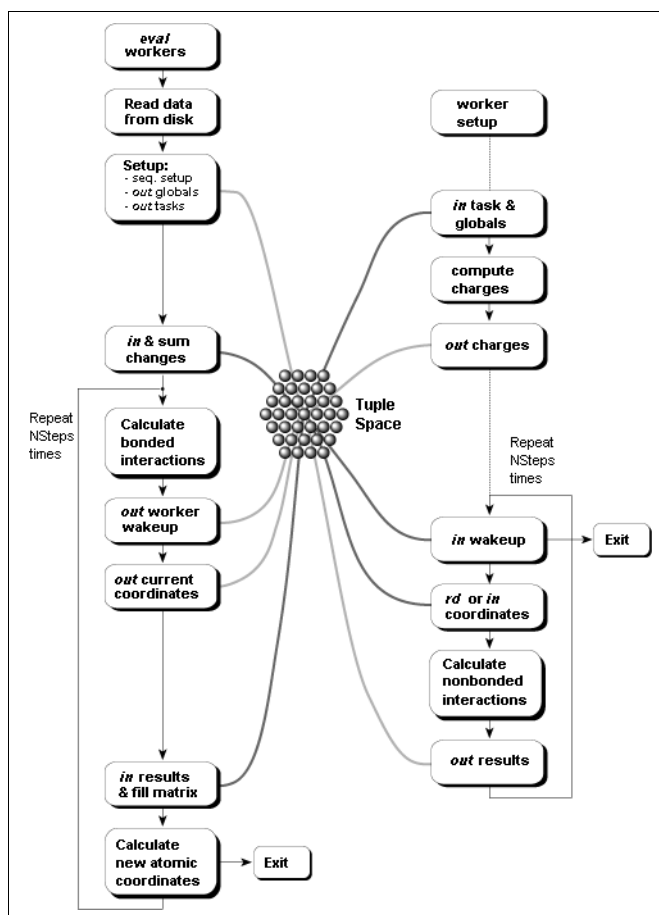
The key routines here are `verl_init`, which performs the setup work, and `verl_step`, which oversees the calculation at each time step. The actual computations are performed by routines called from these functions. As we'll see, it is in the latter that the major changes for the C-Linda version appear.

The following diagram illustrates the structure of the parallel version of the molecular dynamics program. The master still does much of the initial setup work, but one step, summing the special charges, is divided among the workers. Once the master has gathered and processed the workers' results from the setup phase, the main calculation loop begins. Some parts of it remain with the master; in fact, the nonbonded interactions so dominate the execution time that it is not worth parallelizing any of the other steps.

Structure of the Parallel Molecular Dynamics Program

For each loop iteration, the master calculates the bonded interactions and other energy terms, and then divides the work of the nonbonded interactions among the workers, placing the data the worker will need into tuple space. Eventually, the master gathers the workers' results, using them along with the quantities it computed itself to calculate the new position and velocity for each atom in the molecule.

Here is `real_main`, which is a renamed version of the original `main`:

```
real_main(argc,argv)
{
   T = 300.0;
   process_args(argc,argv);
```

```
   /* startup workers */
   for (i = 0; i < num_workers; i++)
       eval("worker", nb_energy_worker(i), num_workers));

   Read data.
   Initialize parameters & data structures.

   verl_init(str,coor,vel,s);
   Te = temperature(str,kinetic_energy(str,vel));
   verl_scale(coor, sqrt(T/Te));
   for (i=0; i < NSteps; i++) {
       verl_step(str, coor, vel, s);
       Te = temperature(str, kinetic_energy(str, vel));
       verl_scale(coor, sqrt(T/Te));
   }

   /* kill workers & clean tuple space */
   for (i = 0; i < num_workers; i++){
       out("wakeup", -1, i);
       in("worker", ?int);
   }
   lexit(0);   /* use C-Linda exit function */
}
```

The only changes here are the two loops which create and terminate the worker processes. The first `for` loop consists of `num_workers` **eval** operations. Each worker is passed a unique integer as its argument, its worker ID number, so to speak. It will use this value to retrieve its own task from tuple space.

The final `for` loop creates one "wakeup" task per worker, with its second field set to -1. This is a poison pill, telling the worker to exit. The loop also retrieves the passive data tuple the worker emits as it dies before generating the next poison pill. This ensures that the master process will not terminate until all the workers have.

If the speed of the shutdown process were important, then the **in** operations could be placed in their own loop, so that all of the poison pills would be generated essentially at once. Then, the workers die in parallel, with the master collecting the "worker" data tuples only after generating all of the poison pills.

In this program, there is no master routine per se; rather, the master functions are split among three routines:

- ◆ `real_main`, which creates and terminates the workers.

- ◆ `nb_setup_energy`, which places global (i.e., non-iteration specific) data into tuple space and controls the parallel special charges calculation.

- ◆ nb_energy, which controls the nonbonded interaction energy calculation for each iteration.

Here is nb_energy_setup:

```
void nb_energy_setup(str,coor,count)
{
    Perform sequential setup work.
    /* global data all workers get once */
    out("A", (str->A):, "B", (str->B):,
        "pack_atc", packatomtc:str->n+1);

    /* taskav=(total work/#workers) = target pairs per worker */
    taskav = (nb_num) * (nb_num-1) / 2 / num_workers;

    /* create worker tasks:
     *     start = beginning of worker's domain
     *     outsize = size of worker's section */
    for (index=1,w_id=0; w_id < num_workers; w_id++){
        start = index;     /* increments of index follow. */

        /* compute start & length for this worker. tasksum holds the
         * number of pairs given to this worker so far. increment it
         * with each successive value of index (= size of current
         * matrix row) until task >= target or we're out of pairs. */
        for (tasksum=0;
                tasksum <= taskav && index < nb_num+1;
                index++)
            tasksum += index;

        if (w_id == num_workers - 1) /* last worker */
            outsize = nb_num + 1 - start;
        else
            outsize = index - start;

        /* start workers' initialization phase */
        out("nbwconfig", wId, start, outsize,
                nb_num, expfac, confac);
        Fill worker-specific data arrays and send to tuple space.
    } /* end worker init loop */

    /* get partial q-sums and nb_list size calculated by workers*/
    qasum = qbsum = 0.0;
    for (i=0 ; i < num_workers; i++){
        in("qsum", ?uasum, ?ubsum, ?ucount);
        qasum += uasum;
        qbsum += ubsum;
        *count += ucount;
    }
```

```
    Finish initialization.
    return;
}
```

After performing the same initial setup steps as the sequential version, the parallel version of `nb_energy_setup` places some global data in tuple space.

The majority of the code shown in this routine is concerned with dividing the work up among the worker processes. The tasks created here define the range of the data each worker is responsible for. This represents a somewhat different technique from the usual master/worker scenario. In the latter, the work is divided into tasks which are independent of any particular worker, and each worker grabs a task when it needs one. Here, the total work is divided among the workers at the beginning of the calculation; the work each worker does is essentially part of its structure—in this case, a function of its worker ID number. This technique of dividing the problem space into discrete chunks and assigning each to one worker is known as *domain decomposition*. The parameters calculated here and communicated to each worker via tuple space will be used in both the summing of special charges done in the setup phase (with results collected at the end of this routine), and in the actual nonbonded interactions computation later in the program.

The scheme for dividing the work here is somewhat complex, but it is designed to ensure good load balancing among the worker processes—this is always a concern when a domain decomposition approach is taken.

For a molecular dynamics calculation, the total number of nonbonded interactions for an N-atom molecule is approximately $N*(N-1)/2$, and so we want each worker to do about

$$(N*(N-1)/2) \ / \ \text{num\_workers}$$

of them. If there are N atoms, there are N^2 possible interactions, but we throw out pairs where an atom is interacting with itself. The factor of $1/2$ comes from the fact that the interactions are symmetric (A's effect on B is equivalent to B's effect on A) and thus only need to be computed once. We should also remove bonded pairs, but if this number is small compared to the total number of interactions, then our formula is a reasonable approximation to the ideal number of interactions per worker.

In some cases, the total work can just be divided as evenly as possible among the workers. Here, we might like to just give each worker the number of interactions we computed above, correcting the last worker's amount for any remainder and other integer rounding we had to do. However, the interaction pairs are actually stored as rows in a lower triangular matrix (a matrix with all of its non-zero elements on or below the diagonal), and for maximal efficiency, we want to give only complete rows of the matrix to each worker. Given this constraint, it is necessary to figure out how many rows to give each worker so that the number of elements each one gets comes out about the same (and close to the target number).

To do so, the program uses the fact that there are I non-zero elements in row I of a lower triangular matrix, and assigns consecutive rows to each successive workers until the number of elements it has exceeds the target number, or until all the rows are gone. This

simple heuristic works quite well so long as the number of workers is much, much smaller than the number of atoms, a condition invariably satisfied by real world molecular dynamics problems.

Once the starting position (in the variable `start`) and the length (`outsize`) have been computed, the "nbwconfig" tuple is sent to tuple space; its second field is `workerid`, the worker's ID number assigned at startup. The tuple's final three arguments are the total number of atom pairs and two constants.

The final `for` loop in the routine gathers the results of the special charges calculations performed in parallel by the workers. The partial results, from the "qsum" tuple, are summed up by `nb_energy_setup` and used in the remaining (sequential) initialization activities, after which the routine returns.

Before continuing with the main thread of the computation, let's look briefly at the beginning of the worker function, `nb_energy_worker`:

```
/* in the configuration for each worker */
in("nbwconfig", workerid, ?start, ?outsize,
   ?nb_n, ?expfac, ?confac);
if (workerid != num_workers-1) {
   rd("A", ?A:, "B", ?B:, "pack_atc", ?a:len);
   out("done_reading");
} else {
   for (rdct = 1; rdct < num_workers; rdct++)
      in("done_reading");
   in("A", ?A:, "B", ?B:, "pack_atc", ?a:len);
}
```

These statements retrieve the "nbwconfig" task tuple and the global values tuple from tuple space. After it starts up, the worker will block until the "nbwconfig" tuple is available. The appearance of this tuple is a trigger that initiates active computation. It is in this sense that we refer to it as a wakeup for the worker, causing it to resume active execution after a significant pause ("sleep").

The `if` statement checks whether this is the worker with the highest worker ID. If not, it **rd**s the globals tuple, and then creates a "done_reading" tuple. If it is the last worker, then it **in**s all the "done_reading" tuples from the other workers and then **in**s the globals tuple, removing it from tuple space. This technique is useful when you want to remove large, unneeded sets of data from tuple space or when some data must be removed because new or updated versions will replace it. We'll see an example of the latter later in the program.

The worker next computes its portion of the special charges sum, and dispatches the results with an **out** operation. It then enters its main infinite `while` loop, performing a few data initialization operations for the coming nonbonded interaction calculation, and then waiting for its next wakeup tuple, again tied to its worker ID and appropriately labelled "wakeup":

```
qasum = qbsum = 0;
Calculate new values.
out("qsum", qasum, qbsum, count);

while(1){
    evdw = elec = esup = 0.0;
    count = 0;
    for(i=0; i <nb_n+1; i++){
        force_update_i = &(force_update[i]);
        force_update_i->x = 0.0;
        force_update_i->y = 0.0;
        force_update_i->z = 0.0;
    }
    in("wakeup", ?wakeflag, workerid);
}
```

Keep in mind that this code executes at the same time as the master routine for this part of the calculation `nb_energy_setup`, which itself waits for the workers to place their partial sums into tuple space.

If we return our focus to the master program thread, once the initialization phase is complete, `real_main` enters its main loop. The routine `verl_step` begins each iteration; eventually, control passes to the routine `nb_energy`. The sequential version of this routine computes the nonbonded interactions; the parallel version of `nb_energy` performs the master functions for this part of the calculation.

`nb_energy` begins by initializing some variables and then sending out the "wakeup" tuple for each worker along with the current coordinates of the atoms in the molecule:

```
void nb_energy(str,coor,force,evdw,elec,esup)
{
    *elec = *evdw = *esup = count = 0.0;

    /*wake up workers */
    for (workerid = 0; workerid < num_workers; workerid++)
        out("wakeup", workerid, workerid);

    /*send out current coordinates on each round */
    out("coor", coor->c:str->n + 1);
```

The worker, who has been waiting to receive the "wakeup" tuple, retrieves it and checks the value in its second field. If this value is not -1, then the next time step commences. The worker next obtains the current coordinates of the atoms:

```
in("wakeup", ?wakeflag, workerid);
if(wakeflag == -1) return 0;/* eat poison and die */
if(workerid != num_workers-1){
    rd("coor", ?c:len);
    out("read_coords");
} else {
```

```
                    for (rdct = 1; rdct < num_workers; rdct++)
                        in("read_coords");
                    in("coor", ?c:len);
                }
```

Most workers **rd** this tuple; however, the last worker waits until all the other workers have **rd** it (using the same technique of gathering semaphore tuples we saw earlier) before removing it from tuple space with the **in** operation. In addition to freeing the memory, this is necessary so that the new coordinates can be unambiguously transmitted to tuple space on the next iteration.

The worker then calculates the nonbonded interactions for its pairs of atoms. It uses code differing only in its loop limits from that in the sequential version of `nb_energy`:

```
    for (i=0, gi=start ; i < outsize; i++, gi++) {
        do many computations...
    }
```

The initial value of the variable `gi` and the limit against which the variable `i` is tested were both obtained from the "nbwconfig" tuple. Once this calculation is complete, the worker sends its results to tuple space:

```
    out("workerdone", fu:start+outsize, elec, evdw, esup, count);
```

It then waits for the next wakeup tuple commencing the next iteration of its `while` loop; eventually, it retrieves a poison pill and exits.

`nb_energy` ultimately gathers the "workerdone" tuples and merges their results into the arrays used by the remainder of the (unaltered sequential) program. The calculation then continues as in the sequential version, with the final calculated energies corrected for temperature at the end of each iteration.

Once all time steps have been competed, `real_main` kills the workers and cleans up tuple space, finishing the master functions it started when the program began. As we have seen, master responsibilities are distributed among three separate routines in this program, each controlling a different phase of the calculation.

5
Using Tuplescope

Tuplescope is an X-based visualization and debugging tool for Linda parallel programs. In addition to the usual debugger features such as single-step mode and breakpoints, Tuplescope can display tuple classes, data in specific tuples, and visual indications of process interaction throughout the course of a program run.

Linda programs are usually debugged using the Code Development System and Tuplescope. This combination offers the most convenient and least intrusive method of debugging. Tuplescope is part of the Linda Code Development System, which simulates parallel program execution in a uniprocessor environment. It requires a workstation running X Windows. This chapter assumes knowledge of standard debuggers and debugging activities. Refer to the manual pages for your system or to the relevant books in the Bibliography for information on these topics.

Program Preparation

Linda programs must be compiled with the **-linda tuple_scope** option to use Tuplescope. The environment variable `LINDA_CLC` or `LINDA_FLC` should also be set to `cds` (for Code Development System). For example, the following commands will prepare the C-Linda program `test24` for use with Tuplescope[†]:

```
% setenv LINDA_CLC cds
% clc -o test24 -linda tuple_scope test24.cl
```

Invoking Tuplescope

Once an executable has been prepared for use with Tuplescope, simply invoking it will start the debugger. For example, the following command would initiate a Tuplescope session with the `test24` application we prepared above:

```
% test24 arguments
```

Of course, this command would need to be run from a shell window within an X Windows session. If desired, you may also include X Toolkit options following the program arguments to customize the resulting Tuplescope windows.

[†] If you use the Bourne shell, the equivalent commands are: `LINDA_CLC=cds; export LINDA_CLC.`

The Tuplescope Display

Below is a canonical diagram of the Tuplescope display (note that exact window properties and look depend on the window manager in use).
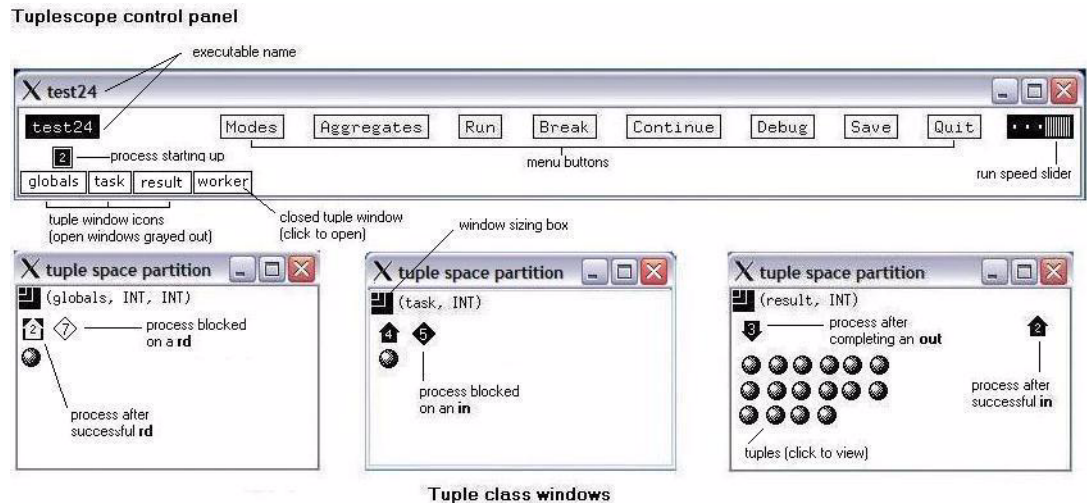
The Tuplescope display consists of two parts: the control panel window and separate windows for each tuple class (tuple classes consist of the compiler-generated distinct partitionings of tuple space, based upon tuple field types and any unique constant strings). Note that the display above is deliberately artificial and was constructed to present all of Tuplescope's features in a compact form rather than to represent any real (or even possible) situation.

The Control Panel

The control panel is a separate Tuplescope window that contains these items:

* The name of the executable, appearing on both the window title bar and in a box in the upper left of the display.

* A row of menu buttons along the top of the window, labeled **Modes** through **Quit**. Click a menu button to display its associated menu (if any) or perform its designated action.

* A slider control bar in the upper right corner of the window that has a sliding bar that moves along a tick-marked area. The default position for the slider is at the far right end of the bar, which represents normal (full) speed. Moving the slider to the left causes program execution to proceed at a slower rate. You can use the slider at any time during program execution to change the execution speed. Slower execution rates represent a middle ground between single -step execution mode, in which execution breaks at every Linda operation, and normal full-speed execution.

◆ A row of buttons for the tuple class window at the bottom left of the control panel. Click on a button to open its corresponding tuple class window. Buttons for currently open tuple class windows are grayed out.

When a process is initiated with an **eval**, an icon for it briefly appears above the tuple class window icons. This icon is a white-outlined black box containing a white number within it; in the diagram, the sample icon contains the number 2. This number functions as a sort of internal process ID and is incremented every time a new process is created; it does not correspond to any user-assigned numbering scheme. Once the process performs an operation on tuple space, this icon disappears and the appropriate icon appears in one of the tuple class windows.

Tuple Class Windows

A tuple class window has the following parts:

◆ A sizing box (black with two white veins), located in the upper left corner of the window controls the size of the window. Click the box to choose minimum, medium, or maximum window size:

Mouse Button	*Effect*
Left	Make the window minimum size
Middle	Make the window medium size
Right	Make the window maximum size

◆ A textual representation of the tuple class, positioned to the right of the sizing box. This shows the tuple class's structure. Here is an example:

```
("globals", INT, INT)
```

Clicking with the left mouse button on the tuple representation string will close the tuple class window.

◆ Spherical icons for each tuple that exists in this class. Left-clicking on a tuple opens a small window that displays its contents. For example:

```
("globals" 1000 225)
```

If a tuple contains a large amount of data, scroll bars will appear, and you can scroll through it, viewing one part of it at a time. You can also scroll with the keyboard arrow keys:

Key	Scrolling Effect
Up arrow	Scroll to previous line
Down arrow	Scroll to next line
Ctrl-up arrow	Scroll to previous page
Ctrl-down arrow	Scroll to next page
Left arrow	Move to the beginning of the tuple
Right arrow	Move to the end of the tuple

If the tuple contains an aggregate, by default the aggregates's contents is not shown; instead, the word BLOCK appears. The Modes and Aggregates menus control the display of aggregates (see <u>Viewing Aggregates</u> below).

Clicking on an individual tuple window with the right mouse button will refresh its contents; clicking on it with the left mouse button will close it.

◆ Icons for processes that have accessed tuples in this class. The form of the icon varies depending on the operation that the process performed and its status. These are the possible icons (all icons display the process number in their center):

Icon Appearance	Meaning
Solid black arrow pointing up	Successful **in** operation
Solid black arrow pointing down	Successful **out** operation
White arrow in black box pointing up	Successful **rd** operation
Solid black diamond	Blocked **in** operation
White diamond	Blocked **rd** operation

There are examples of each type of icon in the Tuplescope display diagram.

Viewing Aggregates

The Display Aggregates item on the Modes menu controls the display of aggregates. Click the Modes button to display its menu, and click the Display Aggregates item to toggle its current state. If the item is selected, a check mark appears to the left of its name. You must select the Exit Modes Menu item to close the modes menu.

When Display Aggregates is on, the Aggregates menu button is active, and you can use its menu to select the data format for subsequent aggregates displays. It contains the choices Long, Short, Float, Double, Character, and Hexadecimal. Only one format choice is active at a given time; a check mark appears to the left of its name. All other formats are grayed out and unavailable. The default format is Long.

To select a different format, first deselect the current format by choosing its menu item. The check mark will then disappear, and the other formats will become active. You may then select the desired format. Exit from this menu by choosing the Exit Aggregates Menu item.

What format a tuple containing an aggregate uses depends on the Dynamic Tuple Fetch setting on the Modes menu. If Dynamic Tuple Fetch is active, then an aggregate is displayed in the current Aggregates menu display format when you click on its tuple icon. If Dynamic Tuple Fetch is not in effect, then the format that was in effect when the tuple entered tuple space will be used regardless of the current setting.

Viewing Process Information

Clicking on a process icon displays a scrollable file window containing the text of the source file corresponding to it. (Tuplescope requires source files to be in the current working directory). The line containing the tuple space operation corresponding to that icon is indicated by a caret ("^"). Scroll bars or the keyboard scrolling keys can be used to examine the source file. To close the window, click the left mouse button anywhere within it.

Tuplescope Run Modes

Tuplescope has a number of different modes for program execution. First, execution speed can be controlled with the slider control on the Tuplescope control panel discussed previously. This is independent of the other run controls we'll look at in this section.

Clicking on the **Run** button will commence program execution. The program will execute in either normal or single step mode, depending on the setting of the **Single Step** item on the Modes menu. When single-step mode is in effect, Tuplescope executes until the next Linda operation takes place. Tuplescope assumes that required source files are in the current directory.

It is not possible to execute a program more than once within a single Tuplescope session. To rerun a program, exit Tuplescope and restart it.

Clicking on the **Break** button will cause program execution to pause at the next Linda operation. Execution will resume when you click on the **Continue** button, which may also be used to resume execution in single step mode.

To exit Tuplescope, click on the **Quit** button.

Using Tuplescope with a Native Debugger

The following method is recommended for combining Tuplescope with a native debugger like **dbx**:

* Compile the program for Tuplescope and the native debugger by including any necessary compiler options on the **clc** command (i.e. **-g** for **dbx**, **-linda tuple_scope** for Tuplescope).

* Execute in single-step mode in Tuplescope until the desired process is created.

* Click on the new process icon with the middle mouse button. This will create a new window running the native debugger attached to that process.

* Set desired breakpoints in the native debugger. Then turn off single step mode in Tuplescope.

* Give the continue execution command to the native debugger to resume execution of that process (e.g. use the **cont** command in **dbx**).

You may now use the native debugger to examine the process. The figure below illustrates a sample combination debugging session.
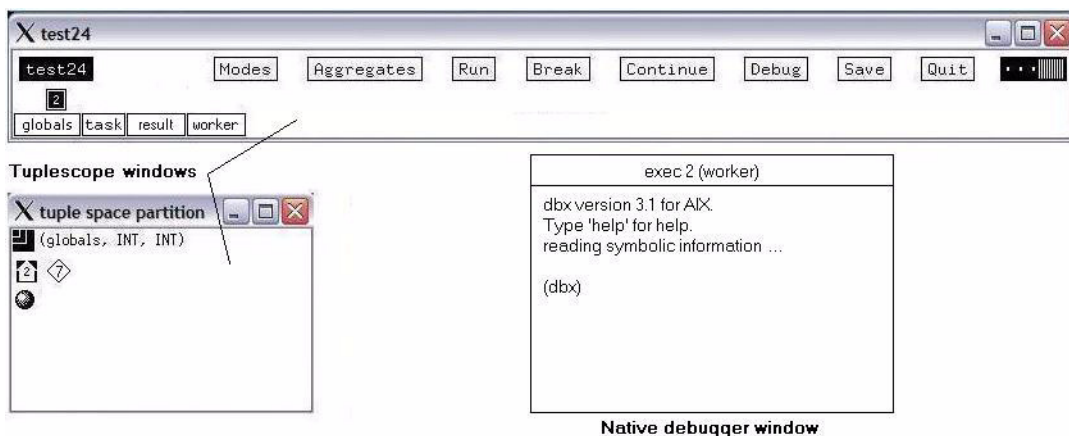
By default, the debugger started is **dbx**, except under HP/UX where it runs **xdb** in an **hpterm** window. A different debugger can be specified by setting the DEBUGGER environment variable to it. If the executable is not in the current path, give the entire pathname as its value; otherwise, its name alone is sufficient. Currently-supported debuggers are **dbx, gdb**, and **xdb** (under HP/UX).

It takes a little practice to understand all the nuances of Tuplescope-native debugger interactions. The trickiest part is usually figuring out where the next continuation command needs to be executed. If the Tuplescope Continue button does not resume execution, try issuing a **next** command to the native debugger process(es).

To exit from a native debugging session without affecting Tuplescope, detach from it before quitting (in **dbx**, give the **detach** command, followed by **quit**). Quitting without detaching first will usually abort the process and cause the Linda program to fail.

It is recommended that you quit from all native debugger processes before exiting from Tuplescope. Pressing the Tuplescope Quit button while debugging windows are still open causes their processes to be terminated "out from under them." Tuplescope will make no attempt to stop the debugger processes, so you will have to do it manually. Some debuggers have difficulty shutting down in this state, so you may have to use the UNIX **kill** command to stop those processes.

The Tuplescope Debugging Language

Clicking on Tuplescope's Debug button brings up a menu that can be used to create, compile, and control debugging programs written in the Tuplescope Debugging Language (TDL). TDL is the means Tuplescope provides for users to specify that certain actions be taken on various program conditions. The various items on this menu have the following meanings:

Item	*Effect*
Edit *program*.debug	Edit the TDL program for this application.
Compile *program*.debug	Compile the TDL program for this application.
Clear Debugging Actions	Cancel all debugging actions in effect.
Exit Debug Menu	Close the Debug menu.

The Edit and Compile items edit and compile the file *program*`.debug` where *program* is the name of the application running under Tuplescope. Edit will open an editor in a separate X window, using the contents of the `EDITOR` environment variable to determine which editor to run.

The Compile item causes the file to be translated into an internal form used by Tuplescope. Successful compilation results in the message `Compilation done`. Otherwise, an error message is printed. Once compiled, the statements in the debugging program go into effect until cancelled by the Clear Debugging Actions item.

TDL Language Syntax

TDL programs consist of one or more lines of the following form:

```
if (condition) then action
```
where *condition* is a test condition, and *action* is some action to be taken when the test is true (1). Note that the parentheses are part of the condition.

Conditions are formed from the following components:

```
[item operator test_value]
```

Note that the brackets are a required part of the syntax.

There are three distinct kinds of conditions:

- Tuple field comparison tests, where *item* is `field` *N* (where *N* is an integer), *operator* is one of the C operators ==, !=, > and <, and *test_value* is a constant. This sort of test selects tuples on the basis of one or more of their fields. For example:

      [field 2 == 2]

  This test chooses tuples whose second field contains the value 2.

  Character strings used as constants must be enclosed in double quotation marks. Single characters must be enclosed in single quotation marks.

  Note that tuples from distinct tuple classes can be selected by the same tuple field comparison. Fields containing aggregates may not be used in such conditions.

- Tuple space operation tests, where *item* is `linda_op`, *operation* is either == or !=, and *test_value* is one of the following: `eval`, `out`, `in`, `rd`, `block_in`, and `block_rd.` This kind of test detects the occurrence of a particular kind of Linda operation. Here is an example:

      [linda_op == eval]

  This condition detects the occurrence of an **eval** operation.

- Process comparison tests, where *item* is `process`, *operation* is one of ==, !=, < and>, and *test_value* is an integer, representing a Tuplescope process number. This sort of test detects when any tuple space operation is performed by any process whose process number fulfills the condition. For example, this condition detects when any process with process number less than 5 accesses tuple space:

      [process < 5]

Multiple conditions can be joined with `and` or `or`, as in this example:

    [process < 5] and [linda_op == out]

The entire composite condition is enclosed in parentheses in the complete TDL statement, as we'll see below.

Action can be one of the following:

Action	*Effect*
break	Pause program execution; resume execution with the Continue button. If a tuple is associated with the tuple space operation that triggers a break, Tuplescope turns it solid black. The process which performed that tuple space operation is marked by a notch in its icon. (Ordinary display of all tuples and process icons is restored when you click the Continue button.)

hide   Suppress display of matching tuples or processes. This can be used to filter out unwanted tuples or processes.

color *color*   Change matching items to the indicated color. *Color* must be on one of: red, orange, yellow, green, blue, indigo, and violet. On monochrome displays, the colors are mapped to distinct black and white pie-shaped icons.

save   Dumps the contents of tuple space to a disk file. This is equivalent to clicking on the Save button during program execution. Save operations are not legal when the Dynamic Tuple Fetch option is in effect.

Here are some examples of complete TDL statements:

```
if ([linda_op == out]) then color red
if ([process < 5] or [process > 7]) then hide
if ([linda_op == out] and [field 2 != 0]) then break
```

The first statement turns the process icons for processes performing **out** operations the color red. The second statement hides all process icons except those for process numbers 5 and 6. The final statement causes execution to pause whenever a tuple is placed in tuple space whose second field is nonzero. Note the syntax of the TDL statements, including both the parentheses and square brackets which must surround conditions.

6

Linda Usage and Syntax Summary

Linda Operations

in(*s*)   Withdraw a tuple matching **s** from tuple space. If no matching tuple is available, execution suspends until one is. If more than one matching tuple exists, one is chosen arbitrarily. When a match is found, the actuals in the matching tuples are assigned to the formals in the corresponding fields of **s**.

rd(*s*)   Look for a tuple matching **s** in tuple space. If a matching tuple is found, actual-to-formal assignment occurs. If no matching tuple exists, the process blocks until one becomes available.

rdp(*s*) & **inp(*s*)**   Predicate forms of **rd** and **in** respectively. They do not block if no matching tuple exists, but return 0/.FALSE. and exit. If a match is found, they return 1/.TRUE. and perform actual-to-formal assignment.

eval(*s*)   Each field of **s** containing a simple function call results in the creation of a new process to evaluate that field. All other fields are evaluated synchronously prior to process creation. When all field values have become available, the tuple **s** is placed into tuple space.

out(*s*)   Synchronously evaluates the fields of the tuple **s** and then places it into tuple space.

You can use the prefix **__linda_** to construct an alternate name for any operation if the shorter name conflicts with other symbols. The prefix begins with two underscore characters.

Formal C-Linda Syntax

```
linda_call : call_type call_body

call_type: in      |      __linda_in    |
           inp     |      __linda_inp   |
           rd      |      __linda_rd    |
           rdp     |      __linda_rdp   |
           out     |      __linda_out   |
           eval    |      __linda_eval  |


call_body: ( element {,element}* )
```

```
element: formal | actual

formal: ? lvalue[:length] | type_name

actual: rvalue[:length]

length: expression

type_name:float | double | struct | union |
   [unsigned] ( int | long | short | char )
```

Timing Functions

The C-Linda function names are listed below.

start_timer()   Initializes and starts the stopwatch in the current process. A separate call to **start_timer** is required in each process where timing is desired.

timer_split(*string***)**
          Takes a stopwatch reading when called and labels the time split with the specified string (length < 32 bytes). The maximum number of timer splits is 32.

print_times()   Prints a table listing all time splits executed so far for this process. Each row includes the time split and its associated string.

Support Functions

The C-Linda function names are listed below. The Fortran-Linda versions have an **f** prepended to the C-Linda function name.

[f]lexit(*status***)**   Replacement for the C **exit** function. The **lexit** routine allows an **eval** process to abort the execution of the routine invoked through the **eval** operation but still continue as an eval server. The *status* value (int) passed to **lexit** is placed into the corresponding field of the live tuple (subject to typecasting restrictions).

[f]lhalt(*status***)**   Terminates execution of the entire Linda application (not just the local process), after calling any termination handlers specified by **lonexit** (see below). Provides the exit value returned by **ntsnet**.

[f]lintoff()   Blocks the interrupts associated with tuple space handling. It is useful for protecting time-consuming system calls from being interrupted. Interrupts should not be disabled for long periods. **linton** and **lintoff** calls can be nested. For many implementations, this is a no op.

[f]linton()   Restores the interrupts associated with tuple space handling (see **lintoff** below). For many implementations, this is a no op.

[f]loffexit(*hd*)   Deletes a specific handler from the list of handlers set up by **lonexit**, where *hd* is the handler descriptor returned by **lonexit**. Returns 0 on success and -1 on failure (descriptor out of range or not referring to an active termination handler).

[f]lonexit(**p,a*)   Names a routine to be called after a Linda process calls **lexit**, **lhalt**, or returns normally. The routine *p* is called as:

```
(*p)(status,a)
```

where *status* is the argument with which **return**, **lexit** or **lhalt** was called, and *a* is typically the address of an argument vector, although it also can be an integer value. Multiple calls can be made to **lonexit**, specifying up to 16 termination handlers, which are called in reverse chronological order (i.e., the last specified routine is called first). **lonexit** returns a unique, non-negative termination handler descriptor upon success or -1 if the termination handler could no be stored.

[f]lprocs()   Returns the total number of processes that have joined the computation (including the master process). In the Code Development System, this function is not meaningful, and it returns the value of the `LINDA_PROCS` environment variable, or the value 6 if it is not defined.
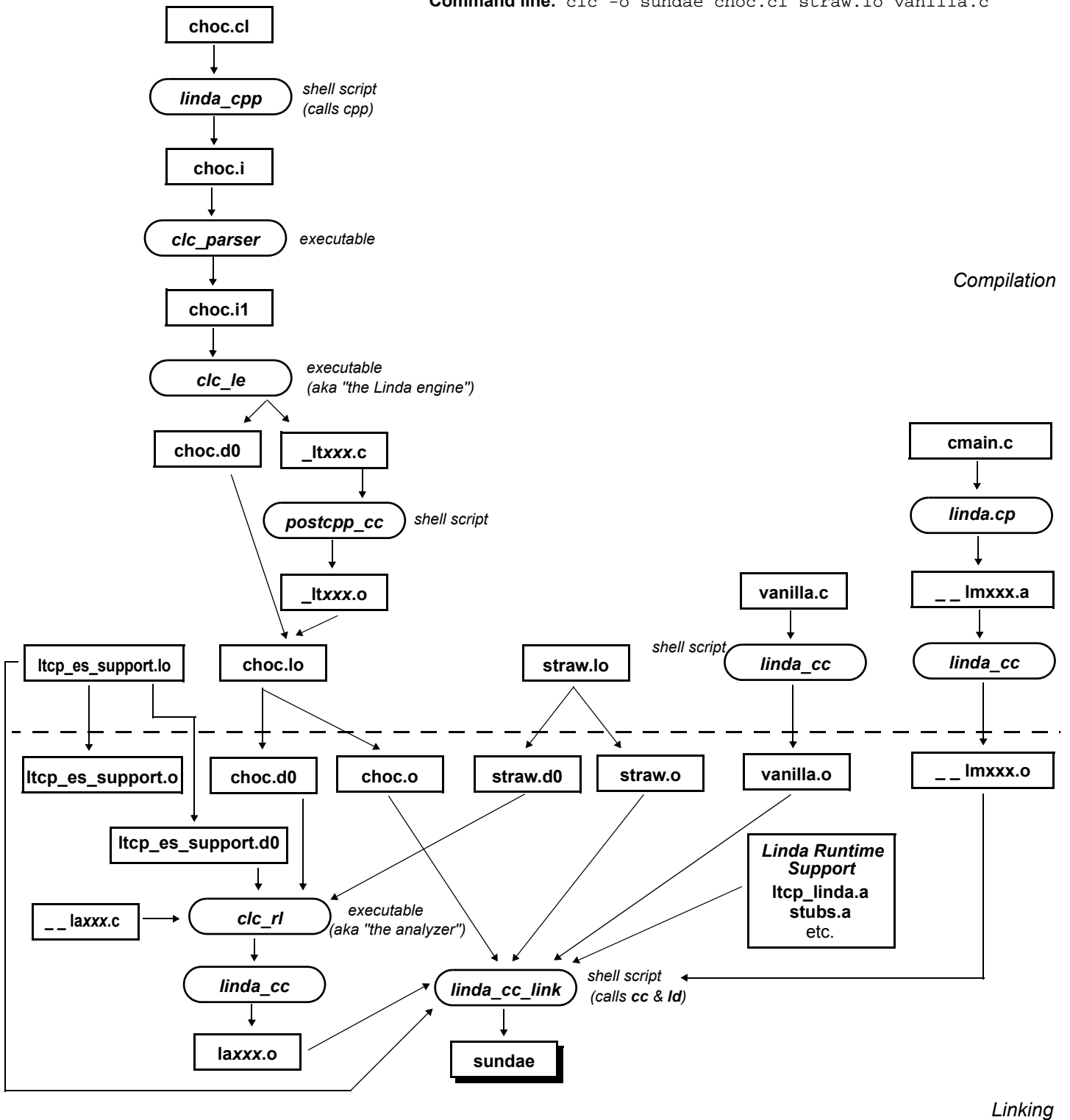
The clc and flc Commands

Command Syntax

```
clc [options]  source files ...
flc [options]  source files ...
```
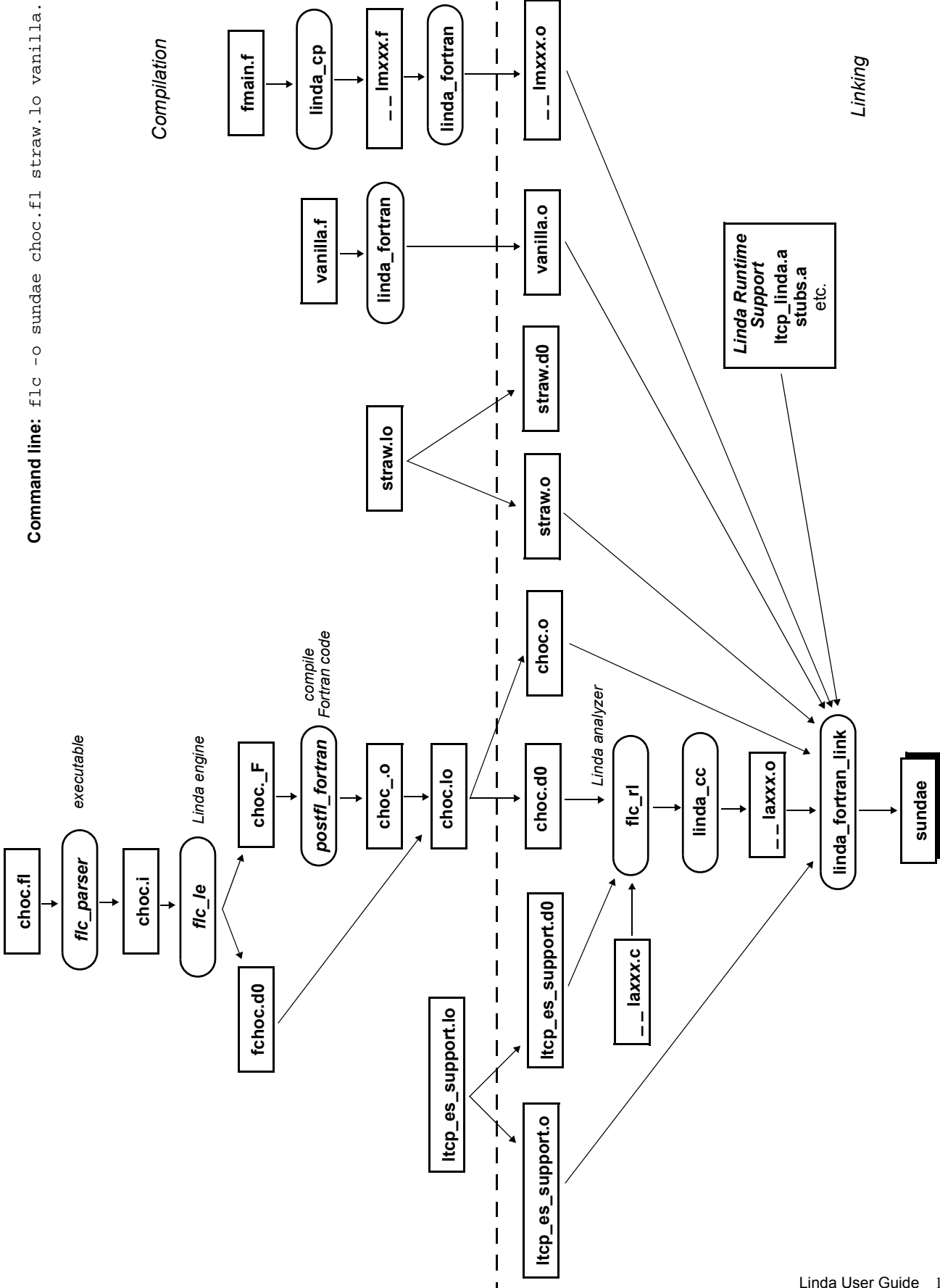
clc expects source files to be of one of the following types: `.c`, `.cl`, `.o`, `.lo`. **flc** expects source files to be of one of the following types: .f, `.fl`, .o, `.lo`. By default, these commands produce the executable `a.out` (override this name with **-o**). The following figure illustrates the compilation and linking process.

Command line: `clc -o sundae choc.cl straw.lo vanilla.c`

choc.cl

linda_cpp *shell script (calls cpp)*

choc.i

clc_parser *executable*

Compilation

choc.i1

clc_le *executable (aka "the Linda engine")*

choc.d0   _lt*xxx*.c

cmain.c

linda.cp

postcpp_cc *shell script*

_lt*xxx*.o

vanilla.c

_ _ lmxxx.a

ltcp_es_support.lo   choc.lo

straw.lo *shell script*

linda_cc

linda_cc

ltcp_es_support.o   choc.d0   choc.o   straw.d0   straw.o   vanilla.o   _ _ lmxxx.o

ltcp_es_support.d0

Linda Runtime Support
ltcp_linda.a
stubs.a
etc.

_ _ la*xxx*.c → clc_rl *executable (aka "the analyzer")*

linda_cc

linda_cc_link *shell script (calls cc & ld)*

la*xxx*.o

sundae

Linking

C-Linda Compiling and Linking Process

Command line: `flc -o sundae choc.fl straw.lo vanilla.f`

Compilation

Linking

fmain.f → linda_cp → __lmxxx.f → linda_fortran → __lmxxx.o

vanilla.f → linda_fortran → vanilla.o

straw.lo → straw.d0
straw.lo → straw.o

Linda Runtime Support
ltcp_linda.a
stubs.a
etc.

choc.fl → *flc_parser* → choc.i → *flc_le*

executable

Linda engine

flc_le → choc.F → *postfl_fortran* → choc_o.o → choc.lo → choc.d0
flc_le → fchoc.d0

compile Fortran code

choc.lo → choc.o

Linda analyzer

choc.d0 → flc_rl → linda_cc → __laxxx.o → linda_fortran_link → sundae

ltcp_es_support.lo → ltcp_es_support.d0
ltcp_es_support.lo → ltcp_es_support.o

__laxxx.c → flc_rl

fchoc.d0 → choc.lo

ltcp_es_support.d0 → flc_rl

choc.o → linda_fortran_link
straw.o → linda_fortran_link
vanilla.o → linda_fortran_link
__lmxxx.o → linda_fortran_link
ltcp_es_support.o → linda_fortran_link

Fortran-Linda Compiling and Linking Process

Command Options

Many of these options have the same meanings as they do with other UNIX compilers.

-c   Suppress linking and only produces `.lo` object files for each Linda source file.

-D*name*[**=***definition*]
               Define a C-preprocessor symbol (**clc** only).

-g   Produce additional symbol table information for debuggers.

-help   Display a help message.

-I*pathname*   Add the specified directory to the include list (**clc** only).

-linda *option* [*arguments*]
               Linda-specific compiler directives. Values for *option* are:

        compile_args *s*   Pass the string *s* on to the native compiler when it is used to compile source files.

        c_args *s*   Pass the string *s* on to the C compiler (**flc** only).

        info   Print out the pathname of the Linda directory and the default size of tuple space.

        link_args *s*   Pass the string *s* on to the compiler when used to link the executable.

        main *file*   Use specified file in place of `cmain.o` or `fmain.o`.

        ts *N*   Initialize tuple space to *N* 200-byte blocks. This option is applicable in Code Development System code.

        tuple_scope   Prepare the object files and/or executable to be run with the Tuplescope debugger. This option can be abbreviated as **t_scope**.

-l*x*   Link in the library `libx.a.`

-L*directory* Add *directory* to the object file/library search list used by the linker.

-o *outputfile*   Name the executable file as indicated.

-v   Display subcommands for each step of the compilation.

-w	Suppress warning messages.
-w72	Suppress warning messages about text beyond column 72 (text is still ignored; **flc** only).

The ntsnet Command

Syntax

ntsnet [*options*] *executable* [*arguments*]

Parameters

options	One or more command line options (listed below). Command line options override configuration file settings.
executable	Executable file to execute.
arguments	Arguments to the executable program.

Options Syntax Convention

When setting boolean resources on the command line, **ntsnet** uses the convention that an option name preceded by a minus sign sets the corresponding resource to true, and one preceded by a plus sign sets the corresponding resource to false.

Command Options

-appl *name*	This option causes **ntsnet** to use *name* as the application name for the purposes of querying the configuration file database. Normally, **ntsnet** uses the executable name, as typed on the **ntsnet** command line, as the application name in the configuration file database. This can be useful if several different executables use the same configuration parameters. Note that **-appl** has no corresponding resource parameter in the configuration file.
-bcast	This option enables the tuple broadcast optimization.
+bcast	This option disables the tuple broadcast optimization. This is the default.
-bcastcache s	This option specifies the size of the broadcast cache in bytes. This size is a trade-off between memory consumption and hit rate. The default size is 1 MB. This resource is only used when **bcast** is true.
-cleanup	This option indicates that remote executables should be removed when execution completes. This is the default. Note that the local executable is protected from removal.
+cleanup	This option indicates that remote executables should not be removed when execution completes.

-d/+d	Synonymous with **-distribute/+distribute**.
-debug	Run application in debug mode (see <u>Debugging TCP Linda Programs</u> on page 69 for more information). This option also changes or overrides the values of several **ntsnet** resources; see the discussion of the **debug** resource later in this chapter for details.
-distribute	This option causes executables to be copied to remote nodes prior to execution. Executables shall only be copied to nodes which are actually going to take part in the execution. After execution completes, **ntsnet** automatically removes the remote executables that it just distributed. The local executable is protected from removal. See the **cleanup** command line option or resource for information on preventing the automatic removal of remote executables.
+distribute	This option indicates that executables are not copied. This is the default.
-fallbackload *load*	
	This option specifies the load average the scheduler shall use for a node if the RPC call to get system load average fails. The default is 0.99. The value specified can be any real number >= 0. If failure of the RPC call indicates that the node is down, this option can be used to set **fallbackload** to a very large value, effectively making the node unavailable to **ntsnet**.
-getload	This option indicates that **ntsnet** should use load average information when scheduling workers on the nodes in the network. This is the default.
+getload	This option indicates that **ntsnet** should not use load average information. This can be used to make worker scheduling consistent between different runs of **ntsnet**. It also makes sense if the **rstatd** daemon is not available on the network.
-h/+h	Synonymous with **-high/+high**.
-help	This option causes **ntsnet** to display the usage message.
-high	This option causes all workers to be run at normal priority and causes Linda internodal communication to run at full speed. This is the default.
+high	This option causes all workers to run at a nice-ed priority (unless specifically overridden on a per node per application basis using the **nice** resource). It also causes Linda internodal communication to be throttled to avoid flooding the network.
-kainterval *seconds*	
	Specifies how often, in seconds, each Linda process sends out a keep alive message. The default is 100 seconds. The range of legal values is 100 to 31536000 (one year). The range is silently enforced. This

resource is only useful when the keep alive mechanism is used (i.e., when **kaon** is true).

-kaon   This option turns on the keep alive mechanism. This is the default.

+kaon   This option turns off the keep alive mechanism.

-loadperiod *minutes*

This option specifies the number of minutes over which the machine load is averaged. Typical values for loadperiod are 1, 5, and 10. The default is 5.

-masterload *load*

This option specifies the load that the master (`real_main`) process is considered to put on the node. The value specified can be any real number $>= 0$. The default is `1`. Typically `1` or some smaller fraction is used. If the master process uses much less CPU time than the workers, the master load should be set smaller than the worker load.

-maxprocspernode *number*

This option specifies the maximum number of Linda processes started on any given node the application is running on. On the local node, **maxprocspernode** includes the master. The default value is `1`.

-mp *number* Synonym for **-maxprocspernode.**

-n *minworkers*[:*maxworkers*]

This option specifies the acceptable range of the number of workers that the application can run with. If *maxworkers* is omitted, it is set to the same value as *minworkers*. **ntsnet** initially starts up the number of workers equal to the maximum of the **minworkers** and **maxworkers** resource values. The master then waits as specified in the **minwait** and **maxwait** resources, for the workers to join the execution group. If at least *minworkers* join before the **maxwait** interval has elapsed, execution shall proceed, otherwise execution shall terminate.

-nodefile *filename*

This option specifies the name of the file containing a list of nodes on which this application can run. The default is `tsnet.nodes`. This resource is for backward compatibility with the old **tsnet** utility. This file is only used if the **nodelist** resource is set to `@nodefile`, which is the default value. See the description of the **nodelist** resource for more details.

-nodelist *"node-specifiers..."*

This option specifies a space-separated list of nodes on which an application may run. This list must be inclosed in quotes if more than one node-specifier is used. A node-specifier can be any one or a combination of the types described below:

  The keyword `@nodefile`

A node name
A user defined resource

See the description of the **nodelist** resource for more details.

Note: If the **-nodelist** option is not used and you have not specifically set the **nodelist** resource in the **ntsnet** configuration file(s), the application will run on the nodes contained in the `tsnet.nodes` file in your current working directory.

-opt "*resource***:** *value***"**

This option specifies a value to override any resource in the configuration file. It provides a mechanism for overriding resources for which no specific command line option is provided.

-p *path*   This option specifies both the directory on a remote node where the Linda executable resides or will be distributed to, and the directory that shall be **cd**ed to prior to executing the remote Linda process. Thus the **-p** option simultaneously overrides both the **rexecdir** and **rworkdir** resources.

Since **-p** specifies a local directory, the value of **-p** is subject to map translation. The translation occurs before the **-p** value overrides the **rexecdir** and **rworkdir** resources. This option is intended to provided a mechanism very similar to the **-p** option on the previous **tsnet** utility.

-redirect   This option turns on the tuple redirection optimization. This is the default.

+redirect   This option turns off the tuple redirection optimization.

-suffix   This option causes a node specific suffix, indicated by the **suffixstring** resource, to be appended to the executable name. This is the default. Note that the default value of **suffixstring** is an empty string.

+suffix   This option indicates that TCP Linda is not to use node-specific suffixes.

-translate   This option indicates that map translation shall be performed. This is the default. Note that it is not an error to have translation on and to have no map file. In that case, no translations will be performed.

+translate   This option indicates that map translation shall not be performed.

-useglobalconfig

This option causes **ntsnet** to use the resource definitions in the global configuration file. The resource definitions in the global configuration file are used in addition to the command line options and the user's local configuration if one exists. This is the default.

+useglobalconfig

This option causes **ntsnet** not to use the resource definitions in the global configuration file. **ntsnet** will only use the command line options and the user's local configuration file if one exists. This is useful if a user does not wish to use the configuration file installed by the system administrator.

-useglobalmap

This option causes **ntsnet** to use the global map translation file. The translations in the global map translation file are used in addition to the user's local map translation file if one exists. This is the default.

+useglobalmap

This option causes **ntsnet** not to use the global map translation file. **ntsnet** will only use the user's local map translation file if one exists. This is useful if a user does not wish to use the map file installed by the system administrator.

-v/+v   Synonymous with **-verbose/+verbose**.

-vv/+vv   Synonymous with **-veryverbose/+veryverbose**.

-verbose   This mode displays remote commands being issued and other information useful for debugging configuration and map files.

+verbose   This option turns off verbose mode. This is the default.

-veryverbose   Turns on very verbose mode, which produces the maximum amount of informational status messages.

+veryverbose   This option turns off very verbose mode. It is the default.

-wait *minwait*[:*maxwait*]

This option specifies the minimum and maximum times to wait for nodes to join the execution group. If *maxwait* is omitted, it is set to the same value as *minwait*. Both default to 30 seconds. Execution will commence once the execution group is set, based on the values of the **minwait**, **maxwait**, **minworkers**, and **maxworkers** resources (see the discussion of these resources in the next section for details).

-workerload *load*

This option specifies the load that a worker will put on a node. The value specified can be any real number $>= 0$. The default is 1. Typically 1 or some smaller fraction is used. A larger value could be used to increase the chances of having one Linda process running on each node.

-workerwait *seconds*

This option specifies the time, in seconds, that a worker waits for a response to its join message, from the master. The default is 90. If a worker does not get a response within the specified time, telling the

worker that it's joined, the worker will exit, and therefore not participate in the application execution.

ntsnet Configuration File Format

This section serves as a reference for the format of both the user (local) **ntsnet** configuration file (`~/.tsnet.config`) and the global **ntsnet** configuration file (`lib/tsnet.config` relative to the Linda tree).

When setting Boolean resources in the configuration files, values can be specified as **true** or **false**, **yes** or **no**, **on** or **off**, or as **1** or **0**.

Resource Definition Syntax

program[*.appl*][*.node*]*.resource*: *value*

where the various components have the following meanings:

`Program` is either the class name `Tsnet`, or a specific instance of this class (i.e., `ntsnet`). In the future, there may be alternate versions of Tsnet-type programs, such as `xtsnet`, but currently there is only `ntsnet`.

`Appl` is either the class name `Appl`, or a specific application name, such as `ping`. The application instance names cannot contain a period; you must convert periods to underscores.

`Node` is the class `Node`, or a specific node name, such as `mysys`. The node instance names can be either the node's official name or a nickname. The node instance names are node names found in either the `/etc/hosts` file, the NIS hosts database, or the Internet domain name database. An example of an official name is, `fugi.mycompany.com`. A typical nickname for this node is `fugi`. If a node name contains a period, you must convert the period to an underscore. The other option would be to use a nickname not containing the "." character.

`Resource` is a variable name recognized by **ntsnet** that can be assigned values.

`Value` is the value assigned to the resource.

If both the `appl` and `node` components are required for a given resource definition, the `appl` component must precede `node`. If an incorrect format is used, the resource definition will be ignored by **ntsnet**.

Resources

Note: All resources are application-specific unless otherwise specified. Also, if the corresponding option is used on the **ntsnet** command line, it takes precedence over the resource value in the configuration files.

available	Specifies whether a node is available for use as a worker. This resource is node-specific. The default is `true`.

bcast   Specifies whether or not the tuple broadcast optimization is enabled. The default is false.

bcastcache   Specifies the size of the broadcast cache. This size is a trade-off between memory consumption and hit rate. The default size is 1Mb. This resource is only used when **bcast** is true.

cleanup   Specifies whether or not remote executables shall be removed from remote nodes after execution completes. Executables are removed only if they were distributed by **ntsnet** in the current execution. The local executable is protected from removal. The default is true.

debug   Specifies whether or not to run in debug mode (see <u>Debugging TCP Linda Programs</u> on page 69 for more information). The default is `false`. If `true`, it also changes the default value for **kaon** to `false`, **workerwait** to `1000000`, and **maxwait** to `1000000`, and overrides the value of **nice** to be `false`.

debugger   Specifies the debugger to use when running in debug mode. The default is **dbx**.

delay   Specifies the delay period in seconds between invocations of **rsh** when **ntsnet** initiates execution on remote nodes. The default value is `0`.

distribute   Specifies whether or not the executable(s) shall be distributed to the remote nodes. Executables are distributed only to those remote nodes that are actually going to take part in the execution. After the execution completes, **ntsnet** automatically removes the remote executables that it just distributed. The local executable is protected from removal. The default is false. See the **cleanup** resource for information on preventing the automatic removal of remote executables.

fallbackload   Specifies the load average the scheduler shall use for a node if the RPC call to get system load average fails. The default is `0.99`. The value specified can be any real number $>=$ `0`. If failure of the RPC call indicates that the node is down, this option can be used to set **fallbackload** to a very large value, effectively making the node unavailable to **ntsnet**.

getload   Specifies whether or not to use load averages when scheduling workers on the nodes in the network. The default is true. This can be used to make worker scheduling consistent between different runs of **ntsnet**. It also makes sense if the **rstatd** daemon is not available on the network.

high   Specifies whether all workers shall run at normal priority and Linda internodal communication should run at full speed. The default is true. If the **high** resource is false, the workers run **nice**'d, unless specifically overridden on a per node per application basis using the **nice** resource (note that **high** being true overrides the setting for **nice**). Also when the **high** resource is false, Linda internodal communication is throttled so

that it does not flood the network and thereby degrade the performance of the network Linda application and other network users. For small networks, 2-4 nodes, specifying **high** as true will probably not make a difference. On large networks, specifying **high** as true, and thus asking the Linda kernel not to throttle internodal communication, may cause the network to flood.

kainterval   Specifies how often, in seconds, each Linda process sends out a keep alive message. The default is `100` seconds. The range of legal values is `100` to `31536000` (one year). The range is silently enforced. This resource is only useful when the keep alive mechanism is used, that is, when **kaon** is true.

kaon   Specifies whether of not the keep alive mechanism is used. The default is `true` unless **debug** is `true`, in which case it is `false`.

lindarcparg   Specifies a string to be passed to the **linda_rcp** shell script called by **ntsnet** to distribute executables to remote nodes. This resource provides a hook enabling the user to change the behavior of the shell script (which can itself be modified by the user). The default implementation of **linda_rcp** (located in the Linda `bin` subdirectory) takes no arguments and so ignores the value of this resource. This is a node-specific resource.

lindarsharg   Specifies a string to be passed to the **linda_rsh** shell script, called by **ntsnet** to start up a worker process on a remote node. This resource provides a hook enabling users to control the behavior of the shell script (which can itself be modified by the user). In the default implementation of **linda_rsh** (located in the Linda `bin` subdirectory), only the string `"on"` is meaningful as a value to this resource. If `"on"` is passed to **linda_rsh**, then the **on** command will be used instead of **rsh** to initiate the remote process. This is a node-specific resource.

loadperiod   Specifies the number of minutes over which the machine load is averaged. This is the load average then used by the worker. Typical values for **loadperiod** are 1, 5, and 10. The default is 5.

masterload   Specifies the load that the master (`real_main`) process is considered to put on the node. The value specified can be any real number >= 0. The default is 1. Typically 1 or some smaller fraction is used. If the master process uses much less CPU time than the workers, then **masterload** should be set smaller than **workerload**.

maxnodes   Specifies the maximum number of nodes on which to execute. The default value is the number of nodes in the node list.

maxprocspernode

Specifies the maximum number of Linda processes started on any given node the application is running on. On the local node, **maxprocspernode** includes the master. The default value is `1`.

maxwait   The maximum amount of time to wait for a valid execution group to be formed. Note that **maxwait** specifies the total time to wait, including the time specified in **minwait**; it does not represent an amount of time to wait over and above the **minwait** interval. The default is 30 seconds, which is the same as the default for **minwait**, unless **debug** is true, when the default value is 1000000 seconds. See the discussion of **minwait** below for more details about this resource.

maxworkers   Specifies the maximum number of workers started for a given application. The default is the number of distinct nodes in **nodelist** minus one for the local node running the master. **ntsnet** initially starts up the number of workers equal to the maximum of the **minworkers** and **maxworkers** resource values. The master then waits the time period specified in the **minwait** and **maxwait** resources for the workers to join the execution group. If at least **minworkers** join within that time, execution shall proceed, otherwise execution shall terminate. See the discussion of **minwait** below for full details.

minwait   Specifies the minimum amount of time to wait to allow an execution group to form in seconds; the default is 30. Execution will proceed according to the following criteria. First, if at any point before the **minwait** interval has elapsed, **maxworkers** workers have joined the execution group, execution will commence at once. When the minwait interval expires, if at least **minworkers** workers have joined the execution group, then execution will begin. Otherwise, execution will begin as soon as **minworkers** workers do join or the **maxwait** interval has expired (the latter includes the time in **minwait**). If there are still not **maxworkers** workers when the **maxwait** interval ends, execution will terminate.

minworkers   Specifies the minimum number of workers started for a given application. The default is 1. Thus, the default minimum shall be a master process and one worker (see **maxworkers**).

nice   Specifies whether or not workers on a specific node run **nice**'d. This resource is node and application specific. The default is true. When the **high** resource is set to true, this resource is ignored. When **debug** is true, its value is overridden to be false.

nodefile   Specifies the pathname of the file containing a list of nodes on which this application can run. The default is `tsnet.nodes`. If **nodefile** and **nodelist** are both undefined, **ntsnet** shall look for the list of nodes on which to run in the `tsnet.nodes` file in the current working directory. This is the default behavior and is backwards compatible with the old **tsnet** utility. The **nodefile** resource is only used if the **nodelist** resource is set to `@nodefile`, which is the default value. See the description of the **nodelist** resource for more details.

nodelist   Specifies a space separated list of nodes on which an application may run. The **nodelist** value can be set to any one or a combination of these items: the key word `@nodefile`, a node name, and user defined

resources. The default is `@nodefile`, plus the local node name. The key word `@nodefile` refers to the **nodefile** resource value, which is a file containing a list of node names. User defined resources provides a way to specify a list of node names symbolically. The user defined resource must be preceded with the indirection symbol. The maximum number of indirections is 16.

redirect	Specifies whether or not tuple redirection optimization is used. The default is true.
rexecdir	Specifies the directory on a remote node where the TCP Linda executable resides. Or if distributing, it also specifies the directory on the remote node where the Linda executable shall be distributed to prior to execution. This resource is node and application specific. The default is the key word `Parallel`. The `Parallel` keyword indicates that **ntsnet** should use the map file to translate the name of the local executable directory for that remote node.
rworkdir	Specifies the remote node's working directory. This resource is node and application specific. The default is the key word `Parallel`. The `Parallel` keyword indicates that **ntsnet** should use the map file to translate the name of the local working directory for that remote node.
speedfactor	Specifies the relative aggregate CPU capability of a particular node. The larger the relative speedfactor, the more capable that particular node is of running multiple workers. This resource is node specific. The default is 1.
suffix	Specifies whether or not to append a node specific suffix, indicated by the **suffixstring** resource, to the executable name. The default is true.
suffixstring	Specifies a suffix to be appended to a particular executable name when run on a particular node (the default is the null string, meaning no suffix). This resource is node and application specific.
threshold	Specifies the maximum load allowed on a specific node. The **ntsnet** scheduler is prevented from starting another worker on this specific node when this **threshold** is reached. This resource is node specific. The default is 20.
translate	Specifies whether map file translation is used. The default is true.
useglobalconfig	Specifies whether the global configuration file is used. The default is true.
useglobalmap	Specifies whether the global map translation file is used. The default is true.

user   Specifies a username to use on remote nodes, instead of your local username. If this resource is unspecified, the remote username is the same as your local username. This resource is node specific.

verbose   Specifies whether or not **ntsnet** works verbosely. The default is false. If the **verbose** resource is true, **ntsnet** displays remote commands being issued, and information about each node specified in the **nodelist** resource.

veryverbose   Specifies whether the maximal amount of status messages should be displayed. The default is false. The **veryverbose** and **verbose** resources are independent.

workerload   Specifies the load that a worker will put on a node. The value specified can be any real number >= 0. The default is 1. Typically 1 or some smaller fraction is used. A larger value could be used to increase the chances of having one Linda process running on each node.

workerwait   Specifies the time, in seconds, that a worker waits for a response to its join message, from the master. The default is 90, unless **debug** is true, in which case it is 1000000. If a worker does not get a response within the specified time, telling the worker that it has joined, the worker will exit, and therefore not participate in the execution of the application.

Map Translation File Format

This section documents the format of the map translation files used by TCP Linda. These files are `.tsnet.map` in the user's home directory—the local map file—and `lib/tsnet.map` (the global map file, located relative to the Linda tree).

Map file entries have one of the following formats:

```
map generic-directory {
   node1 : specific-directory;
   [node2 : specific-directory ;]
   …
   }
mapto generic-directory {
   node1 : specific-directory ;
   [node2 : specific-directory ;]
   …
   }
mapfrom generic-directory {
   node1 : specific-directory ;
   [node2 : specific-directory ;]
   …
   }
```

Note that *generic-directory* need not be a real directory location at all, but can be any string. In this case, the entry has the effect of setting up equivalences among the listed set of remote directories.

Wildcards are allowed in map translation file entries:

- The asterisk character (*) can be used for any node name or as the first component of a node name (e.g. `*.com`).

- The ampersand character (&) substitutes the current node name—at the time and in the context in which the translation is taking place—within a directory pathname. It can be used in either generic or specific directory specifications.

See the discussion in Chapter 4 for full details on map translation file entries.

Environment Variables

The following environment variables are used within the Linda system:

DEBUGGER   Specifies the debugger to use when combining a native debugger with Tuplescope. The default is **dbx** (**xdb** under HP/UX).

LINDA_CC   Used by the `linda_cc` shell script; specifies the C compiler to use for compiling `.c` files (defaults to **cc**).

LINDA_CC_LINK Used by the `linda_cc_link` shell script; specifies the command to use for linking the executable (defaults to **cc**).

LINDA_CLC   Used by the C-Linda compiler; specifies which type of executable to build: `linda_tcp` or `cds` (Code Development System).

LINDA_FLC   Used by the Fortran-Linda compiler; specifies which type of executable to build: `linda_tcp` or `cds` (Code Development System).

LINDA_FORTRAN Used by the `linda_fortran` shell script; specifies the Fortran compiler to use for compiling `.f` files (defaults to **f77** in most cases, and to **xlf** under AIX).

LINDA_FORTRAN_LINK

   Used by the `linda_fortran_link` shell script; specifies the command to use for linking the executable (same defaults as for `LINDA_FORTRAN`**).**

LINDA_PATH   Specifies the path to the Linda installation directory. The directory specification *must* contain a terminal slash.

LINDA_PROCS   Used by the Linda Code Development System as the return value for the **lprocs** and **flprocs** support functions (under CDS, **lprocs** is not truly meaningful and is provided only for compatibility with TCP Linda).

POSTCPP_CC   Used by the `postcpp_cc` shell script; specifies the C compiler to use for compiling `.cl` files (defaults to **cc**).

POSTFL_FORTRAN   Used by the `postfl_fortran` shell script; specifies the Fortran compiler to use for compiling `.f` files generated by Fortran-Linda from `.fl` source files (same defaults as for `LINDA_FORTRAN`).

TSNET_PATH   Used by **ntsnet**; specifies its search path for local executables. Its value is a colon-separated list of directories.

Tuplescope Reference

Menu Buttons

Modes   Set debugging modes on or off.

Aggregates   Specify format for aggregates displays (active when Display Aggregates is on). Available formats are: Long, Short, Float, Double, Character, and Hexadecimal.

Run   Begin program execution.

Break   Pause program execution.

Continue   Resume execution of a paused program.

Debug   Create, edit and/or compile a TDL program.

Save   Save the current contents of tuple space to a file (not available when Dynamic Tuple Fetch mode is in effect). The file is named *program.N*.dump, where program is the application name and *N* is a integer incremented each successive save operation.

Quit   End Tuplescope session.

The Modes Menu

Single Step   Controls whether single step mode is in effect or not (default is off).

Display Aggregates

Controls whether the contents of aggregates are displayed in tuple displays (off by default). If Display Aggregates is not in effect, aggregates in tuple displays appear as the word `Block`. The format for an aggregate display is the one that was in effect when its tuple icon was

opened if Dynamic Tuple Fetch is in effect or when it entered tuple space if Dynamic Tuple Fetch is not in effect.

Dynamic Tuple Fetch

When in effect, tuple contents are copied to Tuplescope only when requested. This mode may speed up execution somewhat, but it has the side effect that not all tuples are always continuously available for inspection as they are under the normal mode.

Reverse Execution

Available in postmortem mode only. Causes execution to run in reverse when in effect.

Exit Modes Menu

Close the Modes menu.

The Debug Menu

Edit *program.debug*

Edit a TDL program named for the current application. The editor specified by the `EDITOR` environment variable is opened in a new window.

Compile *program.debug*

Translate the TDL program to Tuplescope's internal form and put its statements into effect.

Clear Debugging Actions

Cancel all debugging directives in effect via the current TDL program.

Exit Debug Menu

Close the Debug menu.

TDL Language Syntax

TDL statements have the following form:

```
if (condition) then action
```

Conditions have one of the following formats (select one item from each column). This format tests the value in a tuple field and performs the *action* for matching tuples:

```
[ field N ==   constant_value]
          !=
          >
          <
```

This format tests for the specified Linda operation and performs the *action* for matching processes:

```
[  linda_op ==   eval      ]
             !=   out
                  in
                  rd
                  block_in
                  block_rd
```

This format tests for the specified process number and performs the *action* for matching processes:

```
[  process  ==   N     ]
             !=
             >
             <
```

Note that the brackets are part of the condition syntax and must be included. Multiple conditions can be joined with and and or. The entire condition is enclosed in parentheses when it is placed into the TDL statement.

Actions must be one of the following:

break   Pause program execution.

hide   Hide the triggering processes/tuples.

color *c*   Change the triggering process/tuple to the color *c*, one of: red, orange, yellow, green, blue, indigo, and violet.

save   Save the current contents of tuple space to a file, named *program.N*.dump, where program is the application name and *N* is a integer incremented each successive save operation.

7
Sample Programs

This chapter contains five sets of example programs that demonstrate how to use Linda to parallelize serial programs. Each example includes the serial version of a program written in C and a parallel version created with C-Linda. The examples are presented in order of increasing complexity.

Array Assignment

Serial Version

```
/***************************************************************************
* Serial Example - Array Assignment - C Version
* FILE:array.c
* OTHER FILES: make.array.c
* DESCRIPTION:
*   In this simple example, an array is initialized and values assigned.
*   In the parallel version the master task initiates numtasks-1 number of
*   worker tasks.  It then distributes an equal portion of an array to each
*   worker task.  Each worker task receives its portion of the array, and
*   performs a simple value assignment to each of its elements. The value
*   assigned to each element is simply that element's index in the array+1.
*   Each worker task then sends its portion of the array back to the master
*   task.  As the master receives back each portion of the array, selected
*   elements are displayed.
***************************************************************************/

#include <stdio.h>

#defineARRAYSIZE600000


main() {
   int     i; /* loop variable */
   floatdata[ARRAYSIZE]; /* the intial array */

/*********************** initializations *****************************/

   printf("\n*********** Starting Serial Example ***********\n");
   fflush(stdout);

   /* Initialize the array */
   for(i=0; i<ARRAYSIZE; i++)
      data[i] =  0.0;
   /* Do a simple value assignment to each of the array elements */
```

```
   for(i=1; i < ARRAYSIZE; i++)
      data[i] = i + 1;

   printf("Sample results\n");
   printf("   data[1]=%f\n",  data[1]);
   printf("   data[100]=%f\n",  data[100]);
   printf("   data[1000]=%f\n\n",  data[1000]);
   fflush(stdout);

   printf("All Done! \n");
}

###########################################################################
# Serial - Array Makefile
# FILE: make.ser_array.c
# DESCRIPTION:   See ser_array.c
# USE: make -f make.ser_array.c
###########################################################################

CC          =        cc

array: ser_array.c
   ${CC} ser_array.c -o array
```

Parallel Version

```
/**************************************************************************
 *      LINDA Example - Array Assignment - C Version
 * FILE: clinda_array.cl
 * OTHER FILES: make.clinda_array.cl
 * DESCRIPTION:
 *    In this simple example, the master task initiates a number of tasks
 *    indicated by user input. It then distributes portion of the
 *    array into tuple space to be worked on by the workers.
 *
 *    Each worker task retrieves data indicating what portion of the array
 *    the worker should update, and performs a simple value assignment
 *    to each of its elements. The value assigned to each element is
 *    simply that element"s index in the array+1.  Each worker task
 *    then puts its array back into tuple space and will
 *    continue retrieving  work assignments until it is notified to stop
 *
 *    As the master receives back each portion of the array, selected
 *    elements are displayed. When it has received all portions of the array,
 *    It will put out a "poison pill" to notify the workers to stop work.
 **************************************************************************/

#include <stdio.h>
#define ARRAYSIZE 1000
#define MAXWORKERS 20

real_main(int argc, char *argv[]) {
   /*  Local Variables */
```

```
int   numworkers, index, i;
int   data[ARRAYSIZE], result[ARRAYSIZE];
int   extrachunk, id;
int   data_index;
int   chunksize, nbr_chunks;
int   worker();

/*********************** initializations *****************************
* Get the number of tasks from the user.   Then
* define the number of worker tasks and the array partition size as chunksize.
*********************************************************************
*/
numworkers=0;

while ((numworkers > MAXWORKERS) || (numworkers < 1)) {
   printf ("Enter number of workers between 1 and %d\n", MAXWORKERS);
   scanf ("%d", &numworkers);
}

chunksize = 100;

/************************** master task *************************/
printf ("************ Starting LINDA ARRAY Example ***********\n");

/* Initialize the array */

i=0;
while (i < ARRAYSIZE) {
   data[i] =  0;
   i++;
}

/* put it out there */

out ("init_array", data);

/* Start the worker tasks  */

i=1;
while (i <= numworkers) {
   printf ("Starting worker task %d\n", i);
   eval("worker", worker(i));
   i++;
}

nbr_chunks = ARRAYSIZE/chunksize;
i = 1;
index = 0;
while (i <= nbr_chunks) {
   out ("my_section", index, chunksize);
   index = index + chunksize;
   i++;
}/* end while */
```

```
    /* print a few sample values  */

    i=1;
    while (i <= nbr_chunks) {
        in ("result_data", ?id,  ?index, ?result);

        /* put it into the data array */
        data_index = index;
        while (data_index < index+chunksize) {
            data[data_index]=result[data_index];
            data_index++;
        }
        printf ("----------------------------------------------------\n");
        printf ("MASTER: Sample results from worker task %d\n", id);
        printf ("    data( %d ) = %d )\n", index, data[index]);
        printf ("    data( %d ) = %d\n", index+10, data[index+10]);
        printf ("    data( %d ) = %d\n", index+20, data[index+20]);
        printf (" ");
        i++;
    }

    /* now kill off the workers  by telling them there is no more work
    to do, i.e., chunksize = 0 */
    i=1;
    while (i <= numworkers) {
        out("my_section", 0,0);
        i++;
    }

    printf (" MASTER: All Done!\n" );
}

/*********************** worker task *********************************/

int worker(int id) {
    /* local variables */
    int index;
    int data[ARRAYSIZE];
    int chunksize, i;
    int next_index;


    /* Get the initialized array from tuple space  */

    rd ("init_array", ? data);

    /* loop until no more work */

    while (1) {

    /* Index into my portion of the array that was passed in.  Get the chunksize.*/

        in ("my_section",  ?index, ? chunksize);
```

Array Assignment


```
      /* check to see if there is more work */
      if (chunksize == 0 ) {
         /* there is no more work so let other tasks know */
         return(0);
      }

      /* Do a simple value assignment to each of my array elements */

      i=index;
      while (i < index+chunksize) {
         data[i] = i + 1;
         i++;
      }

   /* Send my results back to the master */

   out ("result_data", id, index, data);

   } /* end of loop */

   return (0);
}

#############################################################################
# Linda Simple Array Program Makefile
# FILE: make.clinda_array.cl
# DESCRIPTION: see clinda_array.cl
# USE: make -f make.clinda_array.cl
# Note: To compile using tuple scope use -linda tuple_scope compiler option
#       (CFLAGS = -linda tuple_scope)
#############################################################################
CC        =       clc
CFLAGS    =
array:  clinda_array.cl
   ${CC} ${CFLAGS} clinda_array.cl -o  array
```

Linda User Guide   131

pi Calculation

Serial Version

```c
/* ---------------------------------------------------------------------
 *  Serial pi Calculation - C Version
 *  FILE: ser_pi_calc.c
 *  OTHER FILES: make.pi_calc.c
 *  DESCRIPTION:  pi calculation example program.  C Version.
 *    This program calculates pi using a "dartboard" algorithm.  See
 *    Fox et al.(1988) Solving Problems on Concurrent Processors, vol.1
 *    page 207.  All processes contribute to the calculation, with the
 *    master averaging the values for pi.
 * --------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>

// void srandom (int seed);
double dboard (int darts);

#define DARTS 5000   /* number of throws at dartboard */
#define ROUNDS 10    /* number of times "darts" is iterated */

main() {
   double pi;           /* average of pi after "darts" is thrown */
   double avepi;        /* average pi value for all iterations */
   int i, n;

   srandom (5);
   avepi = 0;
   for (i = 0; i < ROUNDS; i++) {
       /* Perform pi calculation on serial processor */
       pi = dboard(DARTS);

       avepi = ((avepi * i) + pi)/(i + 1);
       printf("   After %3d throws, average value of pi = %10.8f\n",
                 (DARTS * (i + 1)),avepi);
   }
}
/*****************************************************************************
 * dboard
 *****************************************************************************/
#define sqr(x)((x)*(x))
//long random(void);

double dboard(int darts) {
   double x_coord,       /* x coordinate, between -1 and 1  */
         y_coord,       /* y coordinate, between -1 and 1  */
         pi,            /* pi  */
         r;             /* random number between 0 and 1  */
   int score,           /* number of darts that hit circle */
       n;
   unsigned long cconst; /* used to convert integer random number */
                         /* between 0 and 2^31 to double random number */
```

```
                                 /* between 0 and 1  */

   cconst = 2 << (31 - 1);
   score = 0;

/***********************************************************************
 *      Throw darts at board.  Done by generating random numbers
 *      between 0 and 1 and converting them to values for x and y
 *      coordinates and then testing to see if they "land" in
 *      the circle."  If so, score is incremented.  After throwing the
 *      specified number of darts, pi is calculated.  The computed value
 *      of pi is returned as the value of this function, dboard.
 *      Note:  the seed value for rand() is set in pi_calc.
 ***********************************************************************/

   for (n = 1; n <= darts; n++) {
       /* generate random numbers for x and y coordinates */
       r = (double)random()/cconst;
       x_coord = (2.0 * r) - 1.0;
       r = (double)random()/cconst;
       y_coord = (2.0 * r) - 1.0;

       /* if dart lands in circle, increment score */
       if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
          score++;
   }

   /* calculate pi */
   pi = 4.0 * (double)score/(double)darts;
   return(pi);
}


############################################################################
# Serial - pi_calc Makefile
# FILE: make.ser_pi_calc.c
# DESCRIPTION:  See ser_pi_calc.c
# USE: make -f make.ser_pi_calc.c
############################################################################

CC        =       cc

pi_calc: ser_pi_calc.c
   ${CC} ser_pi_calc.c -o pi_calc
```

Parallel Version

```
/* ---------------------------------------------------------------------
 *              LINDA pi Calculation Example - C Version
 *
 * FILE: clinda_pi_calc.cl
 * OTHER FILES: dboard.c, make.clinda_pi_calc.c
 * DESCRIPTION:  Linda pi calculation example program.  C Version.
 *   This program calculates pi using a "dartboard" algorithm.  See
 *   Fox et al.(1988) Solving Problems on Concurrent Processors, vol.1
 *   page 207.  All processes contribute to the calculation, with the
 *   master averaging the values for pi.
 * --------------------------------------------------------------------*/

#include <stdlib.h>
#include <stdio.h>

//void srandom (int seed);
double dboard (int darts);
#define MAXWORKERS 20
#define DARTS 5000   /* number of throws at dartboard */
#define ROUNDS 10    /* number of times "darts" is iterated */

real_main() {
   /* local variables */
   double homepi,       /* value of pi calculated by current task */
          pi,           /* average of pi after "darts" is thrown */
          avepi,        /* average pi value for all iterations */
          pi_recv,      /* pi received from worker */
          pisum;        /* sum of workers pi values */
   int nworkers,         /* number of workers */
   round_nbr,
   wrker, i;
   int worker();

   /* get number of workers */

   nworkers = 0;
   while ((nworkers < 1) || (nworkers > 20)) {
      printf ("Enter number of workers between 1 and 20\n");
      scanf("%d", &nworkers);
   }

   /* start up the workers */

   for (i=1; i <= nworkers; i++) {
      eval ("workers", worker(i));
      printf ("starting worker %d\n", i);
   }

   /* Start collecting the results for each round */
   avepi = 0;
   srandom(0);
   for (round_nbr = 0; round_nbr < ROUNDS; round_nbr++) {
```

```
      pisum = 0;
       /* master does pi calc too */
       homepi = dboard(DARTS);

     /* get results from each worker */
       for (wrker = 0; wrker < nworkers; wrker++) {
           in ("pi_results", round_nbr, ?pi_recv);
           /* keep running total of pi */
           pisum = pisum + pi_recv;
       } /* end for ea worker loop */

       /*  Calculate the average value of pi for this iteration */
       pi = (pisum + homepi)/(nworkers+1); /* +1 includes master's calc */
       /* Master calculates the average value of pi over all iterations */
       avepi = ((avepi * round_nbr) + pi)/(round_nbr + 1);
       printf("   After %3d throws, average value of pi = %10.8f\n",
              (DARTS * (round_nbr + 1)),avepi);
   } /* end for each round loop */

   return (0);
}

/************************************************************************
* Start of Worker Task
************************************************************************/
int worker(int id) {
   /* local variables */
   double my_pi;
   int round;
   /* Set seed for random number generator equal to task ID */

   srandom (id);

   /* Calculate pi using dartboard algorithm */
   for (round = 0; round < ROUNDS; round++) {
      my_pi = dboard(DARTS);
      out ("pi_results", round, my_pi);
   }

   return(0);
}

/************************************************************************
 * dboard
 ************************************************************************/
#define sqr(x)((x)*(x))
//long random(void);

double dboard(int darts) {
    double x_coord,        /* x coordinate, between -1 and 1  */
           y_coord,        /* y coordinate, between -1 and 1  */
           pi,             /* pi  */
           r;              /* random number between 0 and 1  */
    int score,             /* number of darts that hit circle */
```

```
        n;
    unsigned long cconst; /* used to convert integer random number */
                         /* between 0 and 2^31 to double random number */
                         /* between 0 and 1  */

    cconst = 2 << (31 - 1);
    score = 0;

/***********************************************************************
 *     Throw darts at board.  Done by generating random numbers
 *     between 0 and 1 and converting them to values for x and y
 *     coordinates and then testing to see if they "land" in
 *     the circle."  If so, score is incremented.  After throwing the
 *     specified number of darts, pi is calculated.  The computed value
 *     of pi is returned as the value of this function, dboard.
 *     Note:  the seed value for rand() is set in pi_calc.
 ***********************************************************************/

    for (n = 1; n <= darts; n++) {
        /* generate random numbers for x and y coordinates */
        r = (double)random()/cconst;
        x_coord = (2.0 * r) - 1.0;
        r = (double)random()/cconst;
        y_coord = (2.0 * r) - 1.0;

        /* if dart lands in circle, increment score */
        if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
            score++;
    }

    /* calculate pi */
    pi = 4.0 * (double)score/(double)darts;
    return(pi);
}

############################################################################
# Serial - pi_calc Makefile
# FILE: make.ser_pi_calc.c
# DESCRIPTION:  See ser_pi_calc.c
# USE: make -f make.ser_pi_calc.c
# LAST REVISED: 4/18/94 Blaise Barney
############################################################################

CC        =       cc

pi_calc: ser_pi_calc.c
    ${CC} ser_pi_calc.c -o pi_calc
```

Matrix Multiply

Serial Version

```c
/****************************************************************************
* Serial Matrix Multiply - C Version
* FILE: ser_mm.c
* OTHER FILES: make.mm.c
* DESCRIPTION:  Sequential version of a matrix multiply
*   To make this a parallel processing progam this program would be divided into
*    two parts - the master and the worker section.  The master task would
*    distributes a matrix multiply  operation to numtasks-1 worker tasks.
*    NOTE1:  C and Fortran versions of this code differ because of the way
*      arrays are stored/passed.  C arrays are row-major order but Fortran
*      arrays are column-major order.
****************************************************************************
/
#include <stdio.h>

#define NRA 62 /* number of rows in matrix A */
#define NCA 15/* number of columns in matrix A */
#define NCB 7   /* number of columns in matrix B */

main() {
   int    i, j, k;/* misc */
   double a[NRA][NCA], /* matrix A to be multiplied */
          b[NCA][NCB],      /* matrix B to be multiplied */
          c[NRA][NCB];/* result matrix C */

   /* Initialize A, B, and C matrices */
   for (i=0; i<NRA; i++)
       for (j=0; j<NCA; j++)
          a[i][j]= i+j;

   for (i=0; i<NCA; i++)
       for (j=0; j<NCB; j++)
           b[i][j]= i*j;

   for (i=0;i<NRA;i++)
       for (j=0;j<NCB;j++)
           c[i][j] = 0.0;

   /* Perform matrix multiply */
   for (i=0;i<NRA;i++)
       for (j=0;j<NCB;j++)
           for (k=0;k<NCA;k++)
               c[i][j]+= a[i][k] * b[k][j];

   /* Okay, it's a trivial program */
   printf("Here is the result matrix\n");
   for (i=0; i<NRA; i++) {
      printf("\n");
      for (j=0; j<NCB; j++)
          printf("%6.2f   ", c[i][j]);
```

```
   }
   printf ("\n");
}

############################################################################
# Serial - Matrix Multiply Makefile
# FILE: make.ser_mm.c
# DESCRIPTION:  See ser_mm.c
# USE: make -f make.ser_mm.c
############################################################################

CC         =        cc

mm: ser_mm.c
   ${CC} ser_mm.c -o mm
```

Parallel Version

```
/****************************************************************************
*      LINDA Matrix Multiply - C Version
*
* FILE: clinda_mm.cl
* OTHER FILES: make.clinda_mm.cl
* DESCRIPTION:
*   In this code, the master task distributes a matrix multiply
*   operation to n worker tasks.
*   NOTE:  C and Fortran versions of this code differ because of the way
*   arrays are stored/passed.  C arrays are row-major order but Fortran
*   arrays are column-major order.
****************************************************************************/

#include <stdio.h>
#define NRA 62
#define NCA 15
#define NCB 7
#define MAXWORKERS 20

real_main () {
   int    id;
   int    numworkers, nbr_rows, offset,i,j,k;
   int    extra;
   int    worker();   /* worker function */
   int    work_count; /* number of work assignments made */
   double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB], results[NRA][NCB];

   /* ********************** initializations ******************************
    * Get the number of tasks from the user.
    ********************************************************************/
   numworkers=0;

   while ((numworkers > MAXWORKERS) || (numworkers < 1)) {
       printf ("Enter number of workers between 1 and %d\n ", MAXWORKERS);
       scanf ("%d", &numworkers);
```

```
}

/************************* master task************************************/
/*     Initialize A and B  */

for (i=0; i < NRA; i++) {
    for (j=0; j < NCA; j++) {
        a[i][j] = i + j;
    }
}

for (i=0; i < NCA; i++) {
    for (j=0; j < NCB; j++){
        b[i][j] = i * j;
    }
}

/* put out the arrays for the tasks to read */

out("array_b", b);
out("array_a", a);

/*  Put out work for them to do - ten columns at a time */

work_count=0;
nbr_rows =  10;
offset = 0;
extra = NRA % nbr_rows;
if (extra > 0) {
   out ("offset", offset);
   out ("array_data", offset, extra);
   offset = offset + extra;
   work_count++;
} else {
  out ("offset", offset);
}

while (offset < NRA) {
   out ("array_data", offset, nbr_rows );
   offset = offset + nbr_rows;
   work_count++;
}

/* Start up worker tasks  */

id = 1;
while (id <= numworkers) {
   printf ("Starting task %d \n", id);
   eval ("worker", worker());
   id++;
}

/* Receive results from worker tasks */
```

```
    for (k=1;k<=work_count;k++) {
        in ("results", ?offset, ?nbr_rows, ?c);
        /* put results into results array */
        for (i=offset; i<nbr_rows+offset; i++) {
            for (j=0;j<NCB;j++) {
                results[i][j] = c[i][j];
            }
        }
    }

    /* Print results */

    for (i=0;i<NRA;i++) {
        for (j=0;j<NCB;j++) {
        printf ("%.2f  ", results[i][j]);
    }
    printf ("\n");
}


}
/************************** worker task *********************************/

int worker() {
    int offset,rows,length;
    int next_offset;
    int i, j, k;
    double  a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];

    /* First read in the a and b arrays */

    rd ("array_a", ?a);
    rd ("array_b", ?b);

    /* This is the head of an infinite loop over tasks */

    while (1) {

    /* See if there is any work out there to do */
    in ("offset", ?offset);

    if (offset == -1) {
        out("offset", -1);
        return (0);
    }

    /* There is still data to process.   Receive matrix data from master task */

    in ("array_data", offset, ?rows);

    /* Update offset for the next guy */

    if ((offset + rows) < NRA)
        next_offset = offset + rows;
```

```
    else
       next_offset = -1;

    out ("offset", next_offset);


    /* Do matrix multiply */
    for (i=offset;i<rows+offset;i++)
        for (j=0; j<NCB;j++) {
            c[i][j] = 0.0;
            for (k=0;k<NCA;k++)
                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }

    /* Send results back to master task */
    out ("results", offset, rows, c);

    } /* end while loop */

    /* It's all over */

    return(0);
}

##########################################################################
# Linda Matrix Multiply Makefile
# FILE: make.clinda_mm.cl
# DESCRIPTION: see clinda_mm.cl
# USE: make -f make.clinda_mm.cl
# Note: To compile using tuple scope use -linda tuple_scope compiler flags
# (CCFLAGS = -linda tuple_scope)
##########################################################################
#
CC        =       clc
CFLAGS =

mm:      clinda_mm.cl
    ${CC} ${CFLAGS} clinda_mm.cl -o  mm
```

Concurrent Wave Equation

Serial Version

```c
/* -------------------------------------------------------------------------
 * SERIAL Concurrent Wave Equation - C Version
 * FILE: ser_wave.c
 * OTHER FILES: make.wave.c
 * DESCRIPTION:
 *   This program implements the concurrent wave equation described
 *   in Chapter 5 of Fox et al., 1988, Solving Problems on Concurrent
 *   Processors, vol 1.
 *
 *   A vibrating string is decomposed into points.  In the parallel
 *   version each processor is responsible for updating the amplitude
 *   of a number of points over time.
 *
 *   At each iteration, each processor exchanges boundary points with
 *   nearest neighbors.  This version uses low level sends and receives
 *   to exchange boundary points.
 * -------------------------------------------------------------------------*/

#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <time.h>

#define MAXPOINTS 1000
#define MAXSTEPS 10000
#define MINPOINTS 20
#define PI 3.14159265


void init_param(void);
void init_line(void);
void update (void);

int nsteps,                     /* number of time steps */
    tpoints,      /* total points along string */
    rcode;                      /* generic return code */
double values[MAXPOINTS+2], /* values at time t */
       oldval[MAXPOINTS+2], /* values at time (t-dt) */
       newval[MAXPOINTS+2]; /* values at time (t+dt) */

/* -------------------------------------------------------------------------
 * Obtains input values from user
 * ------------------------------------------------------------------------- */

void init_param(void) {
   char tchar[8];

   /* set number of points, number of iterations */
   tpoints = 0;
```

```
   nsteps = 0;
   while ((tpoints < MINPOINTS) || (tpoints > MAXPOINTS)) {
      printf("Enter number of points along vibrating string\n");
      scanf("%s", tchar);
      tpoints = atoi(tchar);
      if ((tpoints < MINPOINTS) || (tpoints > MAXPOINTS))
         printf("enter value between %d and %d\n", MINPOINTS, MAXPOINTS);
   }

   while ((nsteps < 1) || (nsteps > MAXSTEPS)) {
      printf("Enter number of time steps\n");
      scanf("%s", tchar);
      nsteps = atoi(tchar);
     if ((nsteps < 1) || (nsteps > MAXSTEPS))
        printf("enter value between 1 and %d\n", MAXSTEPS);
   }

   printf("points = %d, steps = %d\n", tpoints, nsteps);
}

/*  -------------------------------------------------------------------------
 *      All processes initialize points on line
 *  ---------------------------------------------------------------------- */
void init_line(void) {
   int i, j, k;
   double x, fac;

   /* calculate initial values based on sine curve */

   fac = 2.0 * PI;
   k = 0;
   for (j = 1; j <= tpoints; j++, k++) {
        x = (double)k/(double)(tpoints - 1);
        values[j] = sin (fac * x);
   }
  for (i = 1; i <= tpoints; i++) oldval[i] = values[i];

}

/*  -------------------------------------------------------------------------
 *      Calculate new values using wave equation
 *  ------------------------------------------------------------------------*/
void do_math(int i) {
   double dtime, c, dx, tau, sqtau;

   dtime = 0.3;
   c = 1.0;
   dx = 1.0;
   tau = (c * dtime / dx);
   sqtau = tau * tau;
   newval[i] = (2.0 * values[i]) - oldval[i]
              + (sqtau * (values[i-1] - (2.0 * values[i]) + values[i+1]));
}
```

```c
/*  -------------------------------------------------------------------------
 * Update the points a specified number of times
 * ------------------------------------------------------------------------*/
void update() {
   int i, j, tpts;

   /* update values for each point along string */
   /* update points along line */
  for (j = 1; j <= tpoints; j++) {
      /* global endpoints */
      if ((j == 1) || (j  == tpoints))
         newval[j] = 0.0;
      else
         do_math(j);
  }

  for (j = 1; j <= tpoints; j++) {
      oldval[j] = values[j];
      values[j] = newval[j];
  }


  /* print it out for validation */

  tpts = (tpoints < 10) ? tpoints: 10;
  printf("first %d points (for validation):\n", tpts);
  for (i = 0; i < tpts; i++) printf("%4.2f  ", values[i]);
      printf("\n");
}


/*  -------------------------------------------------------------------------
 * Main program
 * ------------------------------------------------------------------------ */
main() {
   int left, right;

   /* get program parameters and initialize wave values */
   init_param();
   init_line();

   /* update values along the line for nstep time steps */
   update();

   return 0;
}
```

```
##########################################################################
# SERIAL Wave Equation  Makefile
# FILE: make.ser_wave.c
# DESCRIPTION:  See ser_wave.c
# USE: make -f make.ser_wave.c
##########################################################################

CC        =       cc

wave: ser_wave.c
   ${CC} ser_wave.c -o wave -lm
```

Parallel Version

```
/* -------------------------------------------------------------------------
*            Linda Concurrent Wave Equation  - C Example
*
*  FILE: clinda_wave.cl
*  OTHER FILES: draw_wave.c, make.clinda_wave.cl
*
*  DESCRIPTION:
*    This program implements the concurrent wave equation described
*    in Chapter 5 of Fox et al., 1988, Solving Problems on Concurrent
*    Processors, vol 1.
*
*    A vibrating string is decomposed into points.  Each processor is
*    responsible for updating the amplitude of a number of points over
*    time.
*
*    At each iteration, each processor exchanges boundary points with
*    nearest neighbors.  This version demonstrates domain decomposition
*
*   Note: use compile option -lm
*  -------------------------------------------------------------------------
*  -------------------------------------------------------------------------
*  Explanation of constants
*    values(0:1001)   = values at time t
*    oldval(0:1001)   = values at time (t-dt)
*    newval(0:1001)   = values at time (t+dt)
*  -------------------------------------------------------------------------
*/

#include <stdio.h>
#include <math.h>

#define MAXPOINTS 1000
#define MAXSTEPS  10000
#define NWORKERS  4
#define PI 3.14159265

/*     Routine for creating the X graph of the wave */
// extern void draw_wave();

double values[MAXPOINTS+2],newval[MAXPOINTS+2],oldval[MAXPOINTS+2] ;
```

```
real_main () {
   int tpoints; /* total points on the line */
   int nsteps;  /* number of time steps */
   int npts,first,i,k;
   int nmin, nleft;
   int worker();

   tpoints = 0;
   nsteps = 0;

   while ((tpoints < NWORKERS) || (tpoints > MAXPOINTS)) {
      printf("Enter number of points along vibrating string\n");
      scanf("%d",&tpoints);
      if ((tpoints < NWORKERS) || (tpoints > MAXPOINTS)) {
         printf("Enter value between %d and %d\n", NWORKERS, MAXPOINTS);
      }
   } /* end while */

   while ((nsteps < 1) || (nsteps > MAXSTEPS)) {
      printf ("Enter number of times steps\n");
      scanf ("%d", &nsteps);
      if ((nsteps < 1) || (nsteps > MAXSTEPS)) {
         printf("Enter a value between 1 and %d\n", MAXSTEPS);
      }
   } /* end while */

   /* Put out total points, and time steps  */

   out ("globals", tpoints, nsteps, NWORKERS);

   /* Assign chunks of work */

   nmin = tpoints/NWORKERS;
   nleft=tpoints % NWORKERS;

   k = 0;
   for (i = 1; i <= NWORKERS; i++) {
       npts = (i <= nleft) ? nmin +1 : nmin;
       first = k + 1;
       eval ("worker", worker(i,first,npts));
       k=k+npts;
   }

   /* Get results */
   get_results (tpoints,NWORKERS);
}


int worker (int id, int first, int npoints) {
   int left, right;
   int tpoints, nsteps, nwrkers;
   double *ret_val;
```

```
    /* get global values */

    printf("%d: first = %d, npoints = %d\n", id, first, npoints);
    rd ("globals", ?tpoints, ?nsteps, ?nwrkers);

    /* Initialize line  */

    init_line (tpoints, npoints, first);

    /* Determine left and right neighbors */

    if (id == nwrkers)
       right = 1;
    else
       right = id + 1;

    if (id == 1)
       left = nwrkers;
    else
       left = id - 1;

    /*  Update the line */
    update(left,right,tpoints,nsteps,npoints,first,id);

    /* Output the results */

    /* the first value ([0]) was only used for passing data to
       neigbors so reset array to send back only values we are interested in */

    ret_val = values+1;
    out ("results", id,first, ret_val:npoints);

}

/*  ----------------------------------------------------------------------
 *     Initialize points on line
 *  ----------------------------------------------------------------------
*/

int init_line (int tpoints, int npoints, int first) {
    /* local variables */
    int i, j, k;
    double x, fac;

    /* Calculate initial values based on sine curve */
    fac = 2.0 * PI;
    k = first-1;
    for (j=1; j <= npoints; j++, k++) {
        x = (double)k/(double)(tpoints - 1);
        values[j] = sin (fac * x);
    }

    for (i=1; i<= npoints; i++) {
        oldval[i] = values[i];
```

```
    }
}

/*   -------------------------------------------------------------------------
 *      Update all values along line a specified number of times
 *   -------------------------------------------------------------------------
 */

int update(int left, int right, int tpoints, int nsteps, int npoints,
           int first, int id) {
   /* local variables */
   double dtime, c, dx, tau, sqtau;
   int iteration, i, j;

   /* Update values along the wave for nstep time steps */
   dtime = 0.3;
   c = 1.0;
   dx = 1.0;
   tau = (c * dtime / dx);
   sqtau = tau * tau;

   /* Update values for each point along string */

   for (i=1; i<= nsteps; i++) {
       iteration = i;
       /* Exchange data with "left-hand" neighbor */
       if (first != 1) {
          out("RtoL",left, iteration, values[1]);
          in("LtoR",id, iteration, ?values[0]);
       } /* end if */

       /* Exchange data with "right-hand" neighbor */
       if (first+npoints-1 != tpoints) {
          out("LtoR",right,iteration, values[npoints]);
          in("RtoL",id,iteration, ?values[npoints+1]);
       } /* end if */

       /* Update points along line */
       for (j=1; j <= npoints; j++) {
           /* Global endpoints */
           if ((first+j-1 == 1) || (first+j-1 == tpoints)) {
              newval[j] = 0.0;
           }
           else {
              /* Use wave equation to update points */
              newval[j] = (2.0 * values[j]) - oldval[j]
                  + (sqtau * (values[j-1] - (2.0 * values[j])
                  + values[j+1]));
           } /* end if */
       } /* end npoints for */

       for (j=1; j <= npoints; j++) {
           oldval[j] = values[j];
           values[j] = newval[j];
```

```
      }
   } /* end nsteps for loop */

   return(0);
}


/*  ----------------------------------------------------------------------
 *     Receive results from workers and print
 *  ----------------------------------------------------------------------
 */
int get_results (int tpoints, int nworkers) {
   /* local variables */
   int i, j, k, tpts, first, length;
   double results[1000];

   /* Store worker's results in results array */
   for (i=1; i <= nworkers; i++) {
       /* Receive number of points and first point */
       in ("results", i, ?first,?results:length);
       /* put it into the results array */
       first = first - 1; /* adjust first to array starting at 0 */
       j=0;
       for (k=first; k < length + first; k++, j++) {
           values[k] = results[j];
       }
   }

   tpts = (tpoints < 10) ? tpoints : 10;

   /* Print something out for validation */

   printf ("first  %d points (for validation): \n", tpts);
   for (i = 0; i < tpts; i++) printf ("%4.2f  ", values[i]);
   printf ("\n");

   /* Display results with draw_wave routine */

   // draw_wave (&values);

   return (0);
}
```

```
############################################################################
# Linda Concurrent Wave Equation Program Makefile
# FILE: make.clinda_wave.cl
# DESCRIPTION: see clinda_wave.cl
# USE: make -f make.clinda_wave.cl
# Note: To compile using tuple scope use -linda tuple_scope compiler option
#       (CFLAGS = -lm -linda tuple_scope)
############################################################################
#
CC        =        clc
CFLAGS = -lm


wave:       clinda_wave.cl
   ${CC} ${CFLAGS} clinda_wave.cl -o  wave
```

2D Heat Equation

Serial Version

```
/****************************************************************************
 * Serial HEAT2D Example - C Version
 * FILE: heat2D.c
 * OTHER FILES:
 * DESCRIPTIONS:  This example is based on a simplified
 * two-dimensional heat equation domain decomposition.  The initial
 * temperature is computed to be high in the middle of the domain and
 * zero at the boundaries.  The boundaries are held at zero throughout
 * the simulation.  During the time-stepping, an array containing two
 * domains is used; these domains alternate between old data and new data.
 ****************************************************************************/

#include <stdio.h>
#define NXPROB 11
#define NYPROB 11
struct Parms
{
  float cx;
  float cy;
  int nts;
} parms = {0.1, 0.1, 50};

main() {
   float u[2][NXPROB][NYPROB];
   int ix, iy, iz, it;
   void inidat(), prtdat(), update();

   /*********************************************************************
   **  Initialize grid.
   *********************************************************************/
   inidat(NXPROB, NYPROB, u);
```

```
   prtdat(NXPROB, NYPROB, u, "initial.dat");
   for (ix = 0; ix <= NXPROB-1; ix++) {
       u[1][ix][0] = u[0][ix][0];
       u[1][ix][NYPROB-1] = u[0][ix][NYPROB-1];
   }

   for (iy = 0; iy <= NYPROB-1; iy++) {
       u[1][0][iy] = u[0][0][iy];
       u[1][NXPROB-1][iy] = u[0][NXPROB-1][iy];
   }

   /**********************************************************************
   **   Iterate over all timesteps.
   **********************************************************************/
   iz = 0;
   for (it = 1; it <= parms.nts; it++) {
       update(NXPROB, NYPROB, &u[iz][0][0], &u[1-iz][0][0]);
       iz = 1 - iz;
   }

   prtdat(NXPROB, NYPROB, &u[iz][0][0], "final.dat");
}

/*****************************************************************************
 *  subroutine update
 *****************************************************************************/
void update(int nx, int ny, float *u1, float *u2) {
   int ix, iy;

   for (ix = 1; ix <= nx-2; ix++) {
       for (iy = 1; iy <= ny-2; iy++) {
           *(u2+ix*ny+iy) = *(u1+ix*ny+iy)  +
             parms.cx * (*(u1+(ix+1)*ny+iy) + *(u1+(ix-1)*ny+iy) -
               2.0 * *(u1+ix*ny+iy)                      ) +
             parms.cy * (*(u1+ix*ny+iy+1) + *(u1+ix*ny+iy-1) -
               2.0 * *(u1+ix*ny+iy)                   );
       }
   }
}

/*****************************************************************************
 *  subroutine inidat

 *****************************************************************************/
void inidat(int nx, int ny, float *u1) {
   int ix, iy;

   for (ix = 0; ix <= nx-1; ix++) {
       for (iy = 0; iy <= ny-1; iy++) {
           *(u1+ix*ny+iy) = (float)(ix * (nx - ix - 1) * iy * (ny - iy - 1));
       }
   }
}
```

```
/*************************************************************************
 * subroutine prtdat
 *************************************************************************/
void prtdat(int nx, int ny, float *u1, char *fnam) {
   int ix, iy;
   FILE *fp;

   fp = fopen(fnam, "w");
   for (iy = ny-1; iy >= 0; iy--) {
       for (ix = 0; ix <= nx-1; ix++) {
           fprintf(fp, "%8.3f", *(u1+ix*ny+iy));
           if (ix != nx-1)
               fprintf(fp, " ");
           else
               fprintf(fp, "\n");
       }
   }
   fclose(fp);
}

#############################################################################
# Serial - heat2D Makefile
# FILE: make.ser_heat2D.c
# DESCRIPTION:  See ser_heat2D.c
# USE: make -f make.ser_heat2D.c
#############################################################################

CC         =       cc

heat2D: ser_heat2D.c
   ${CC} ser_heat2D.c -o heat2D
```

Parallel Version

```
/*************************************************************************
*               LINDA Example - Heat Eqaution Domain Decomposition - C Version
*
* FILE: clinda_heat2d.cl
* OTHER FILES: make.clinda_heat2d.cl
* DESCRIPTION:
*
* This example is based on a simplified two-dimensional heat equation
* domain decomposition.  The initial temperature is computed to
* be high in the middle of the domain and zero at the boundaries.  The
* boundaries are held at zero throughout the simulation.  During the
* time-stepping, an array containing two domains is used; these domains
* alternate between old data and new data.
*
* The parallel version decomposes the domain into tiles, and each tile is
* advanced in time by a separate process.  The master process performs as a
* worker also.  NXPROB, NYPROB, and NXTILE may be varied to study performance;
* note that the code assumes that NXTILE evenly divides (NXPROB-2).  This
* quotient is the number of parallel processes.  In the C version, the
```

```
* tiles are actually strips.  This is because C-Linda does not have the rich
* Fortran-90 array section notation that is available in Fortran-Linda.
*
* Two data files are produced: an initial data set and a final data set.
* An X graphic of these two states displays after all calculations have
* completed.
***************************************************************************/
#include <stdio.h>
#define NXPROB 11
#define NYPROB 11
#define NXTILE 3
struct Parms
{
  float cx;
  float cy;
  int nts;
} parms = {0.1, 0.1, 50};

real_main() {
    float u[NXPROB][NYPROB];
    int nwrkers, ntile, ixmin, ixmax, idum, i, ix, iz, it, worker();
    void inidat(), prtdat();

    /*
    **  Initialize grid.
    */
   printf("Grid size: X= %d  Y= %d  Time steps= %d\n",NXPROB,NYPROB,parms.nts);
    printf("Initializing grid and writing initial.dat file...\n");
    inidat(NXPROB, NYPROB, u);

    prtdat(NXPROB, NYPROB, u, "initial.dat");

    /*
    ** Compute tile extents, putting grid data into Tuplespace.
    ** Create workers for all but the right-most tile.
    */
    out("struct parms", parms);
    nwrkers = 1;
    ntile = 0;
    for (ix = 1; ix <= NXPROB-2; ix += NXTILE) {
       ixmin = ix;
       ixmax = ix + NXTILE - 1;
       ntile++;
      if (ixmax < NXPROB-2) {
         /* start up the workers */
         printf ("starting worker %d\n", nwrkers);
         eval("worker", worker(ixmin, ixmax));
         nwrkers++;
      }
      /* put out the initial data into tuple space */
      out("initial data", ixmin, &u[ixmin][0]:NXTILE*NYPROB);
    }

    /*
```

```
   ** Put grid boundary data into Tuplespace.
   */
   out("left", &u[0][0]:NYPROB);
   out("right", &u[NXPROB-1][0]:NYPROB);

   /*
   ** Call a worker to compute values for right-most tile.
   */
   idum = worker(ixmin, ixmax);

   /*
   ** Gather results from Tuplespace.
   */
   for (i = 1; i <= ntile; i++) {
      in("result id", ?ixmin, ?ixmax);
      in("result", ixmin, ?&u[ixmin][0]:);
   }

   printf("Writing final.dat file and generating graph...\n");
   prtdat(NXPROB, NYPROB, u, "final.dat");
}

/********************************************************************
 * Worker routine
 ********************************************************************/
int worker(int ixmin, int ixmax) {

   float utile[2][NXTILE+2][NYPROB];
   int ix, iy, iz, it;
   void step();

   /*
   ** Get parameters and initial data from Tuplespace.
   */
   rd("struct parms", ?parms);
   in("initial data", ixmin, ?&utile[0][1][0]:);

   /*
   ** Set edges of utile to grid boundary values if appropriate.
   */
   if (ixmin == 1) {
      in(  "left", ?&utile[0][0][0]:);
      for (iy = 0; iy <= NYPROB-1; iy++) {
         utile[1][0][iy] = utile[0][0][iy];
      }
   }

   if (ixmax == NXPROB - 2) {
      in( "right", ?&utile[0][NXTILE+1][0]:);
      for (iy = 0; iy <= NYPROB-1; iy++) {
         utile[1][NXTILE+1][iy] = utile[0][NXTILE+1][iy];
      }
   }
```

```
   for (ix = 0; ix <= NXTILE+1; ix++) {
      utile[1][ix][0] = utile[0][ix][0];
      utile[1][ix][NYPROB-1] = utile[0][ix][NYPROB-1];
   }

   /*
   **  Iterate over all timesteps.
   */
   iz = 0;
   for (it = 1; it <= parms.nts; it++) {
      step(it-1, ixmin, ixmax, &utile[iz][0][0], &utile[1-iz][0][0]);
      iz = 1 - iz;
   }

   /*
   ** Put results into Tuplespace.
   */
   out("result id", ixmin, ixmax);
   out("result", ixmin, &utile[iz][1][0]:NXTILE*NYPROB);
}

/***********************************************************************
 * step
 **********************************************************************/
void step(int it, int ixmin, int ixmax, float u1[NXTILE+2][NYPROB],
          float u2[NXTILE+2][NYPROB]) {
void update();

   /*
   ** Put boundary data into Tuplespace.
   */
   if (ixmin != 1) {
      out( "west", it, ixmin, &u1[      1][0]:NYPROB);
   }

   if (ixmax != NXPROB-2) {
      out( "east", it, ixmax, &u1[NXTILE][0]:NYPROB);
   }

   /*
   ** Get boundary data from Tuplespace.
   */
   if (ixmin != 1) {
      in( "east", it, ixmin-1, ?&u1[        0][0]:);
   }

   if (ixmax != NXPROB-2) {
      in( "west", it, ixmax+1, ?&u1[NXTILE+1][0]:);
   }

   /*
   ** Update solution.
   */
   update(NXTILE+2, NYPROB, u1, u2);
```

```
}

/**********************************************************************
 * update
 **********************************************************************/
void update(int nx, int ny, float *u1, float *u2) {
   int ix, iy;

   for (ix = 1; ix <= nx-2; ix++) {
      for (iy = 1; iy <= ny-2; iy++) {
         *(u2+ix*ny+iy) = *(u1+ix*ny+iy)  +
            parms.cx * (*(u1+(ix+1)*ny+iy) + *(u1+(ix-1)*ny+iy) -
               2.0 * *(u1+ix*ny+iy)                        ) +
            parms.cy * (*(u1+ix*ny+iy+1) + *(u1+ix*ny+iy-1) -
               2.0 * *(u1+ix*ny+iy)                     );
      }
   }
}

/**********************************************************************
 * inidat
 **********************************************************************/
void inidat(int nx, int ny, float *u1) {

   int ix, iy;

   for (ix = 0; ix <= nx-1; ix++) {
      for (iy = 0; iy <= ny-1; iy++) {
         *(u1+ix*ny+iy) = (float)(ix * (nx - ix - 1) * iy * (ny - iy - 1));
      }
   }
}

/**********************************************************************
 * prtdat
 **********************************************************************/
void prtdat(int nx, int ny, float *u1, char *fnam) {

   int ix, iy;
   FILE *fp;

   fp = fopen(fnam, "w");
   for (iy = ny-1; iy >= 0; iy--) {
      for (ix = 0; ix <= nx-1; ix++) {
         fprintf(fp, "%8.3f", *(u1+ix*ny+iy));
         if (ix != nx-1)
            fprintf(fp, " ");
         else
            fprintf(fp, "\n");
      }
   }
   fclose(fp);
}
```

```
###########################################################################
# FILE: make.clinda_heat2D.cl
# DESCRIPTION:  see clinda_heat2D.cl
# USE: make -f make.clinda_heat2D.cl
# Note: To compile using tuple scope use -linda tuple_scope compiler option
#       (CFLAGS = -linda tuple_scope)
###########################################################################

CC      =       clc
OBJ     =       heat2D
SRC     =       clinda_heat2D.cl
XLIBS   =
CFLAGS =

${OBJ}: ${SRC}
   ${CC} ${CFLAGS} ${SRC} ${INCLUDE} ${LIBS} ${XLIBS} -o ${OBJ}
```

Bibliography

Nicholas Carriero and David Gelernter. **How to Write Parallel Programs: A First Course**. The MIT Press, Cambridge, MA, 1990.

An introduction to developing parallel algorithms and programs. The book uses Linda as the parallel programming environment.

David Gelernter and David Kaminsky, "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha," **Proceedings of the ACM International Conference on Supercomputing**, July 19-23, 1992.

An overview of and preliminary performance results for Piranha.

Brian W. Kernighan and Dennis M. Ritchie. **The C Programming Language**. Second Edition. Prentice-Hall, Englewood Cliffs, NJ, 1988.

The canonical book on C.

Robert Bjornson, Craig Kolb and Andrew Sherman. "Ray Tracing with Network Linda." **SIAM News** 24:1 (January 1991).

Discusses the Rayshade program used as an example in Chapter 3 of this manual.

Harry Dolan, Robert Bjornson and Leigh Cagan. "A Parallel CFD Study on UNIX Networks and Multiprocessors." **SCIENTIFIC Technical Report N1**.

Discusses the Freewake program used as an example in Chapter 2 of this manual.

Mark A. Shifman, Andreas Windemuth, Klaus Schulten and Perry L. Miller. "Molecular Dynamics Simulation on a Network of Workstations Using a Machine-Independent Parallel Programming Language." **SCIENTIFIC Technical Report N2**.

Discusses the Molecular Dynamics code used as an example in Chapter 3 of this manual.

Mike Loukides. **UNIX for FORTRAN Programmers**. Sebastopol, CA: O'Reilly & Associates, 1990.

*Chapter 5 discusses the use of UNIX debugging utilities, including **dbx**. Chapter 8 provides a good overview of standard UNIX profiling utilities.*

Tim O'Reilly et. al. **X Window System User's Manual**. Volume 3 of *The X Window System* series. Sebastopol, CA: O'Reilly & Associates, 1988-1992.

Chapter 10 of the standard edition and Chapter 9 of the Motif edition discuss the theory/ philosophy of X resources, which has been used in the design of the Network Linda configuration files.

Index